Noname manuscript No.
(will be inserted by the editor)

Efficient and Scalable k-Means on GPUs

Clemens Lutz · Sebastian Breß · Tilmann Rabl · Steffen Zeuch · Volker Markl

2018-07-31

Abstract *k*-Means is a versatile clustering algorithm widely used in practice. To cluster large data sets, state-of-the-art implementations use GPUs to shorten the data to knowledge time. These implementations commonly assign points on a GPU and update centroids on a CPU.

We identify two main shortcomings of this approach. First, it requires expensive data exchange between processors when switching between the two processing steps point assignment and centroid update. Second, even when processing both steps of *k*-means on the same processor, points still need to be read two times within an iteration, leading to inefficient use of memory bandwidth.

In this paper, we present a novel approach for centroid update that allows us to efficiently process both phases of *k*-means on GPUs. We fuse point assignment and centroid update to execute one iteration with a single pass over the points. Our evaluation shows that our *k*-means approach scales to very large data sets. Overall, we achieve up to $20\times$ higher throughput compared to the state-of-the-art approach.

Clemens Lutz
DFKI GmbH
E-mail: clemens.lutz@dfki.de

Sebastian Breß
DFKI GmbH
E-mail: sebastian.bress@dfki.de

Tilmann Rabl
TU Berlin
E-mail: rabl@tu-berlin.de

Steffen Zeuch
DFKI GmbH
E-mail: steffen.zeuch@dfki.de

Volker Markl
TU Berlin
E-mail: volker.markl@tu-berlin.de

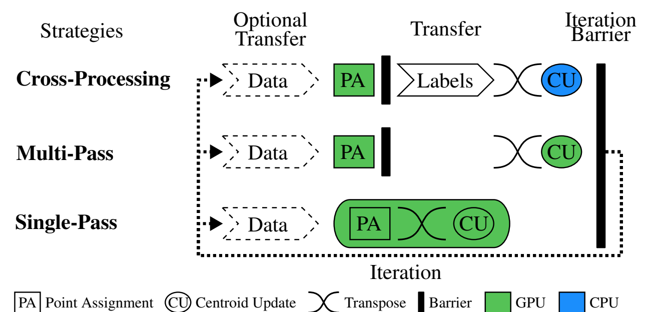


Fig. 1 *k*-Means Execution Strategies.

1 Introduction

In diverse fields of study, practitioners apply the *k*-means algorithm [25, 27] to find patterns in large data sets. Especially in the area of big data, applications such as genome data analysis [19, 41, 43] and climatology [8, 10, 22] require high-performance *k*-means implementations for a short data to knowledge time. Furthermore, many algorithms build on *k*-means to cover new use cases, e.g., BIRCH [44] and streaming *k*-means [38]. Thus, speeding up *k*-means enables data scientists to obtain new insights by exploiting larger data sets in higher quality. At the same time, GPUs have become significantly faster and cheaper, promising fast execution of machine learning algorithms. However, previous research on GPU-accelerated relational databases [5, 6, 15, 17, 18, 21, 34, 35] has shown that algorithms require careful design and tuning to fully exploit GPUs.

In Figure 1, we optimize *k*-means for GPUs, step by step. The *Cross-Processing* strategy results from the two phases in each iteration of *k*-means: point assignment and centroid update. Research over the last decade focused mostly on accelerating these phases on CPU or GPU separately [7, 9, 13, 16, 37, 42]. In particular, they compute the point assignment phase on the GPU and perform the centroid update on the CPU. This split between CPU and GPU causes

the *cross-processing problem*, because the split requires a data exchange over the PCI-e bus in each iteration. In contrast, some approaches avoid the split by performing point assignment and centroid update on the GPU [4, 24]. However, both types of approaches introduce artificial synchronization barriers between both phases. The barriers require these approaches to make two passes over the point data. The resulting *multi-pass problem* decreases the throughput up to a factor of two. Overall, current approaches either do not process the centroid update on the GPU or do it in a way that cannot efficiently fuse both phases.

In this paper, we address both problems to fully exploit modern GPU's processing power for *k*-means. Our contributions are as follows [1]:

1. We introduce a novel centroid update algorithm for GPUs that solves the cross-processing problem by eliminating artificial synchronization barriers between phases.
2. We introduce the Single-Pass execution strategy on GPUs to solve the multi-pass problem by making a single pass over the data points.
3. We show how *k*-means scales to very large data sets that exceed the GPU's dedicated memory.
4. We evaluate the Multi-Pass and Single-Pass strategies against the state-of-the-art Cross-Processing strategy on CPUs and GPUs in scenarios which are either data-intensive or compute-intensive.

The remainder of the paper is structured as follows. In Section 2, we highlight the differences between CPUs and GPUs, and briefly explain the *k*-means algorithm. After that, we outline our GPU-optimized centroid update in Section 3. Then, we contribute our novel *Single-Pass* execution strategy in Section 4 and show its application for arbitrary large data sets in Section 5. In Section 6, we present our experimental results. Finally, we review related work in Section 7 and conclude in Section 8.

2 Background

In this section, we present the state-of-the-art knowledge for executing *k*-means on modern hardware. First, we highlight the general differences between GPU and CPU execution in Section 2.1. After that, we introduce *k*-means and its two-phase execution pattern in Section 2.2.

2.1 CPU vs. GPU Execution

CPUs and GPUs differ significantly in their architecture and execution model. On the one hand, CPUs are optimized for single-thread performance using few complex cores and large caches. On the other hand, GPUs are optimized for data-parallel processing using many simple cores and small caches. Due to these significant differences, algorithms need to be optimized for the underlying architecture.

The architecture of modern GPUs is organized in tens of compute units. A compute unit physically runs 32 threads in a batch [31]. Logically, multiple batches inside the same compute unit form a *work group*, and all threads in a work group execute code in a lock-step manner. There are two methods available to synchronize threads. First, *fine-grained* synchronization maintains a barrier within a work group, which all threads must reach before execution continues. In contrast, *coarse-grained* synchronization requires that all threads terminate and restart their GPU program.

To manage data, GPUs provide a memory hierarchy containing *global memory*, *local memory*, and multiple layers of caches. Global memory is shared among all work groups inside the GPU. In general, threads in a work group *coalesce* adjacent accesses to global memory to a single operation. Modern GPUs provide up to 32 GB capacity, which is cached in a global L2-cache (1.5-4MB) and a per-compute unit L1-cache (16-32KB). In contrast, local memory is disjoint from global memory, and is shared among all threads in the same work group. Local memory provides a capacity between 32-64KB and faster access compared to global memory. Unlike the hardware-controlled L1 and L2 caches, accesses to local memory are explicitly managed in software.

Compared to CPUs, GPUs introduce a different work scheduling strategy. In particular, GPUs are able to schedule multiple work groups simultaneously. Thus, multiple work groups use distinct partitions in local memory, which potentially reduces performance. While all modern GPU processors support atomic memory instructions for global memory, only some support them for local memory. In contrast, CPUs rely heavily on single-thread performance and out-of-order execution. Furthermore, they provide larger caches but a lower degree of parallelism. Overall, GPUs and CPUs have different strengths and weaknesses. To exploit their capabilities efficiently, an algorithm has to take their individual differences into account to achieve peak performance.

2.2 *k*-Means Algorithm

k-Means [25, 27] represents a popular algorithm for cluster analysis in modern machine learning applications. It partitions N observations into k clusters such that each observation belongs to the cluster with the nearest mean. The input to *k*-means is a set of *points* in an $\mathbb{R}^d$ space spanned by d *features*, and a parameter k that specifies the number of clusters. *k*-Means produces a *centroid* per cluster and a *label* per point as output. A centroid defines the mean of the points forming a cluster, while a label identifies the cluster to which a point belongs.

The actual processing of *k*-means is performed in two phases [3]. First, during the *point assignment phase*, a point

[1] We previously sketched our work as a two-page short paper [26].

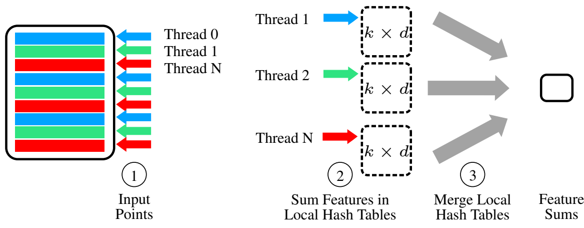


Fig. 2 Feature Sum strategy: Cluster Merge. Each color represents a single thread processing one point at a time.

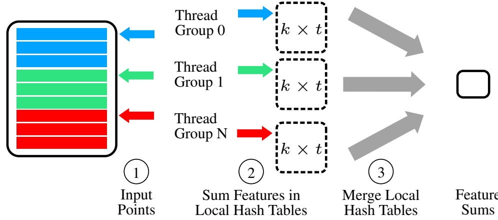


Fig. 3 Feature Sum strategy: Partitioned Features. Each color represents a thread group processing multiple points at a time.

is assigned to the cluster of its nearest centroid. The nearest centroid is determined by the Euclidean distance. Second, during the *centroid update phase*, the means are recalculated for each cluster. Both phases together form one k-means iteration that is repeated either until the mean squared error of the old and new centroids converges below an ϵ or a defined iteration limit is exceeded.

3 Efficient Centroid Update

In this section, we discuss different strategies to compute the centroid update efficiently on GPUs. Our efficient GPU implementation allows us to perform k-means completely on the GPU, which eliminates the labels transfer over PCI-e required by Cross-Processing. In general, centroid update consists of two phases: first, computing the *Feature Sum* and second, the *Mass Sum*. We discuss these individually in Sections 3.1 and 3.2, as they require different approaches. The new centroids are obtained by dividing feature sum vectors by the mass sum vector.

3.1 Feature Sum

During the Feature Sum subphase, k-means adds up the individual feature values of all points that belong to the same cluster and stores the result in a vector of feature sums. As a result, Feature Sum is logically equivalent to a SQL group-by aggregation query, where we group by the centroid IDs and sum each feature of the points separately. We discuss the relation to group-by in Section 7.

Cluster Merge. We now describe the design of Cluster Merge, depicted in Figure 2. The points are stored in column major format, such that each feature of a point is stored in a separate array. Together, the arrays represent a matrix.

In Cluster Merge, each thread processes a point (i.e., all features, see ① in Figure 2) and aggregates the point in its local hash table (see ② in Figure 2) using the point's label (cluster ID) as key. In a final step, all threads merge their partial results with a reduction [14] to the final feature sum vector (see ③ in Figure 2).

Cluster Merge allocates thread-local memory per thread to store feature sums for each cluster (i.e., the working set). This has the benefit that no atomic writes or other thread synchronization mechanisms slow down performance when scaling the number of threads.

However, Cluster Merge requires a large amount of cache space per thread. Each thread requires $4 \times k \times d$ bytes of additional cache space to store its working set. Thus, we can either scale the number of threads and face cache thrashing, or we use too few threads and, as a result, underutilize the GPU processors and memory controllers.

Partitioned Features. Our *Partitioned Features* strategy, shown in Figure 3, optimizes Feature Sum for the hardware architecture of GPUs. The key idea of Partitioned Features is to exchange data through *workgroup-based synchronization*. Each thread is responsible for a particular feature instead of a complete data point. Conceptually, each thread in the same workgroup processes a different feature of the same data point. Depending on the number of features, a workgroup processes multiple points at once. This allows us to reduce the working set (and thus cache thrashing) of each thread to $4 \times k$ bytes (compared to $4 \times k \times d$ for Cluster Merge). This approach requires a barrier within the work group, which is generally fast on GPUs because threads execute in lockstep.

The key differences to Cluster Merge are as follows. First, Partitioned Features shares a local hash table *per workgroup* (see ② in Figure 3). Each workgroup consists of t threads, where each thread stores one sum per centroid (k sums in total). Thus, each hash table has a size of $k \times t$ elements. Since each thread writes exclusively to its own k sums within the shared hash table, no atomic writes are needed. Second, each thread of the same workgroup reads one feature of the same point and adds it to the feature sum of the cluster indicated by the label. The whole workgroup processes a horizontal partition of the point data (see ① in Figure 3). This way, a single hash table per workgroup is sufficient to store the intermediate result and accesses to the point data can be coalesced. As in Cluster Merge, each point's label is read from global memory only once. Sharing the label among threads introduces a barrier within the work group. In a final step, all work groups reduce their partial results to obtain the final feature sum vector (see ③ in Figure 3). Depending on the data set, the number of features d can be less than or greater than the number of threads per

work group (i.e., $d < t$ or $d > t$). We handle these cases by either processing multiple points in the same work group ($d < t$, depicted in Figure 3), or by partitioning features of the same point over multiple work groups ($d > t$).

In conclusion, our Partitioned Features strategy achieves better cache-efficiency than Cluster Merge through workgroup-based synchronization. Thus, Partitioned Features runs with a higher number of threads on GPUs compared to Cluster Merge, which results in more efficient GPU execution.

3.2 Mass Sum

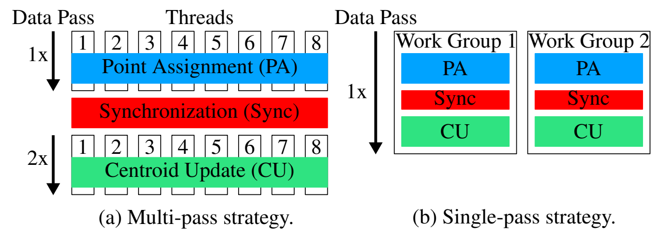
Mass Sum is the second subphase in centroid update. It calculates a histogram which counts the number of points for every cluster. In contrast to a general histogram computation, Mass Sum uses the point label as index to directly access a bucket, instead of first calculating the bucket's index. Three differences distinguish Mass Sum from Feature Sum: a) it has only one dimension, b) the labels are counted, thus no point data is read, and c) it increments integers instead of adding floating point numbers. The last property is relevant because GPUs are faster at atomic integer increments than atomic floating point additions.

Depending on k, different approaches are needed to balance synchronization cost between threads and merging cost for combining intermediate results. Thus, we consider four strategies for Mass Sum: Global Atomic, Partitioned Global, Partitioned Local, and Partitioned Private.

Global Atomic. The simplest way to compute a histogram on a GPU is to use one global histogram that is updated by all threads. We refer to this approach as Global Atomic, because all threads need to synchronize globally on each bucket using atomics to ensure a correct result. Global Atomic is the most simple strategy and performs well for a large number of clusters (>1000). However, for a small number of clusters, it causes heavy contention. For these cases, we need a different strategy.

Partitioned Global (Local). We provide each work group of threads a dedicated histogram stored in global memory to reduce contention between threads (*Partitioned Global*). This way, threads in different workgroups do not need to synchronize, which significantly improves performance. This comes at the cost of merging the individual histograms at the end of the computation by a parallel reduction step. If the hardware supports atomic additions in local memory [32], we store the histograms in GPU local memory (*Partitioned Local*). Without hardware support, we fall back to Partitioned Global.

Partitioned Private. Partitioned Private provides each thread with its own copy of the histogram residing in local memory. Partitioned Private incurs no contention over buckets in case of a small number of clusters, in contrast to the Partitioned Local strategy. Note that Partitioned Private incurs higher merging overhead than the previous strategies and also requires more space in local memory. This strategy outperforms the other approaches if k is small.



(a) Multi-pass strategy. (b) Single-pass strategy.

Fig. 4 Fusing point assignment and centroid update by synchronizing threads per workgroup.

To summarize, a Multi-Pass strategy for the GPU requires an efficient centroid update on GPUs. In our analysis, we separate centroid update into logically distinct subphases, Feature Sum and Mass Sum. In Feature Sum, we address the large working set of Cluster Merge by introducing the space-efficient Partitioned Features strategy. For Mass Sum, we note that Partitioned Private and Partitioned Local operate in processor cache, but have different trade-offs. With these improvements, centroid update is a cache-efficient, Multi-Pass k-means strategy for GPUs.

4 Single-pass GPU k-Means

All state-of-the-art k-means algorithms on GPUs compute point assignment and centroid update in two separate phases. During each phase, the point data is read from global memory, which leads to inefficient use of memory bandwidth. In this section, we show how both phases can be combined to process one iteration with a single pass over the point data. We illustrate the difference between single and multi-pass execution in Figure 4.

Fusing point assignment and centroid update is not trivial due to the data layout necessary for per-thread accesses. In point assignment, threads access individual points (rows of data). In contrast, during centroid update, threads access individual features of points (columns of data). To fuse these phases, we need to overcome the differences in data layout. Our key idea is to cache data in local memory and transpose it on-the-fly from a row-oriented to a column-oriented format. We need to solve two challenges: transposing the data layout of the points and efficiently synchronize threads.

Transposing the Data Layout. Transposing data in cache requires sufficient space in local memory to store one point per thread (i.e., $4 \times t \times d$ bytes, with t threads). Additionally, there must be enough space left for the working set of the centroid update algorithm. The Partitioned Features strategy is essential for this key idea to work on GPUs: in contrast to Cluster Merge, the working set of Partitioned Features is very small. Thus, we use the remaining space to

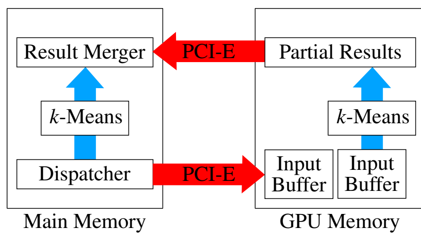


Fig. 5 Support large data sets by streaming data to the GPU in chunks.

transpose the data in local memory. Concretely, the Cluster Merge strategy with Partitioned Private (Mass Sum) uses $4t(kd + k + d)$ bytes of local memory, whereas the Partitioned Features strategy uses $4t(2k+d+1)$ bytes, assuming t threads per work group.

Block-wise Synchronization. Threads in Partitioned Features write to distinct features of the new centroids, but, because of the transpose, each thread processes multiple points. Thus, each thread reads labels computed by other threads. Our solution is that threads exchange labels only within a work group. To this purpose, we synchronize the work group with a thread barrier between point assignment and centroid update, depicted in Figure 4(b).

In summary, the key to fuse point assignment and centroid update is to use the Partitioned Features strategy (Feature Sum) and Partitioned Private strategy (Mass Sum) as the centroid update algorithm. Only then do we have enough cache space to efficiently transpose the layout of point data.

5 Supporting Very Large Data Sets

In this section, we extend our GPU-based execution strategies to handle data sets larger than GPU global memory. The key idea is to divide the point data into chunks, such that at least two chunks fit into global memory (Figure 5). Then, a dispatcher selects and transfers chunks to global memory via PCI-e. In parallel to ongoing transfers, the GPU computes a k-means iteration for each chunk, and adds the output to a partial result. This partial result resides in global memory and consists of partial sums and counts per cluster, as previously described in Feature Sum and Mass Sum. Optionally, chunks are also processed on CPU. In this case, as multiple processors are used, chunks residing in the memory attached to a processor can be processed in-place using operator placement [5]. After all chunks are processed, the partial results of the CPU and the GPU are merged by summing and dividing results to obtain new centroids.

In contrast to previous chunking approaches [24, 40], we aggregate results of chunks locally on the GPU, instead of transferring each individual result to main memory over PCI-e. Thus, we merge in parallel on the GPU and reduce transfer overhead.

6 Evaluation

In this section, we evaluate our k-means strategies. We review our setup in Section 6.1. In Section 6.2, we present our results, and discuss them in Section 6.3.

6.1 Setup and Configuration

First, we introduce our methodology and experimental setup. After that, we describe our applied hardware tuning settings and data set. Finally, we introduce the experiments that we use to evaluate our strategies.

Methodology and Environment. We evaluate our implementation on a CPU and a GPU. Our implementation is based on OpenCL and we measure runtime of OpenCL kernels. CPU experiments are conducted on an Intel Core i7-6700K (“Skylake”) at 3.4 GHz with 4 cores (8 hyperthreads) and 32 GB RAM. GPU experiments use an Nvidia GeForce GTX 1080 (“Pascal”) with 8 GB memory. The test machine runs Ubuntu 16.04 LTS and we use Intel OpenCL Runtime 16.1.1 for CPU code, and Nvidia OpenCL 1.2 (CUDA 8.0.0) for GPU code. Unless stated otherwise, on the GPU we exclude data transfer time to dedicated memory to avoid biased observations of execution time.

Hardware Tuning. To reach peak performance on each processor, we evaluate every OpenCL kernel with different tuning settings. Tuning settings include caching in local memory (disabled on CPUs to avoid memcopy), memory access patterns, vector lengths, and work group sizes. We format data in a column-oriented layout.

Data Sets. Our measurements use synthetically generated data sets following Arthur and Vassilvitskii [3]. Thus, we first sample 10 centroids using a uniform random distribution in a hypercube, where each dimension ranges from -100 to 100. Then, for each centroid, we sample an equal number of points from a normal distribution in a radius of 10 around the centroid. We measure throughput on 2 GB data sets with varying k and d values. The 2 GB data set size amortizes runtime system overheads while minimizing experiment runtime. We set the number of features as $d \in \{2, 4, 8, \dots, 256\}$ and adjust the number of points N such that data size remains constant.

Experiments. We conduct eight experiments that investigate scalability of parameters on GPUs. The first and second experiment are microbenchmarks to determine the best strategies for Feature Sum and Mass Sum. The third experiment compares different k-means strategies and breaks down execution times for each strategy. Experiments four and five examine the scaling behavior of each strategy while varying either the k or the d values. We transfer varying chunk sizes over PCI-e in experiment six. Experiment seven investigates scalability for datasets exceeding the memory capacity of the GPU. The last experiment evaluates the cost-effectiveness of a GPU-based k-means and a distributed

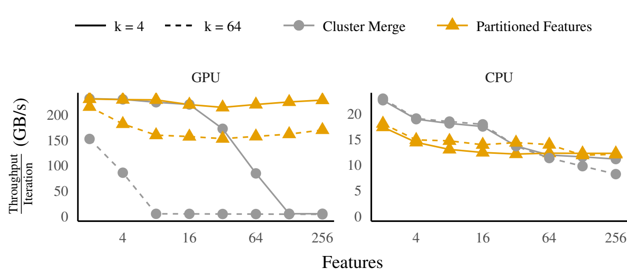


Fig. 6 Comparison of Feature Sum strategies.

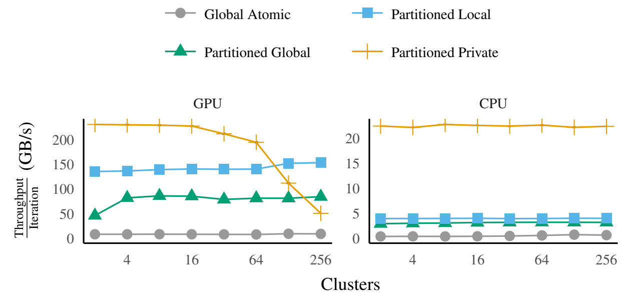


Fig. 7 Comparison of Mass Sum strategies.

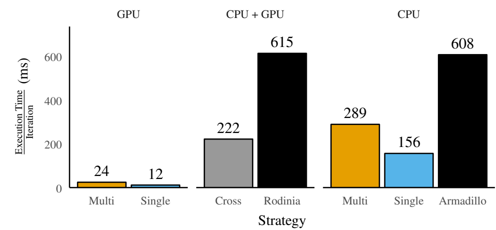


Fig. 8 Comparison of execution times between k-means strategies on CPU and GPU for 4 features and 4 clusters.

CPU-based k-means. We choose these experiments because they explore the parameter space and analyze the trade-offs of different strategies. We report mean and standard deviation (if above 5%) over 30 iterations.

Baselines. We compare our results to two open-source frameworks, Armadillo [36] version 8.400 on the CPU and Rodinia [9] version 3.1 on the GPU. Armadillo is a linear algebra library for C++. It uses a Single-Pass k-means strategy on the CPU with OpenMP multi-threading support. We run 8 threads. Rodinia features a CUDA k-means benchmark that uses the Cross-Processing strategy. Due to limited hardware texture memory and lack of data streaming support, Rodinia is constrained to 512 MB data size. We extrapolate results to 2 GB. In addition to Rodinia, we have implemented our own Cross-Processing strategy optimized for our hardware.

6.2 Results

In this section, we present our evaluation results.

6.2.1 Feature Sum Strategies

We compare two strategies to compute Feature Sum (Figure 6): the Cluster Merge proposed by Li et al. [24] and our Partitioned Features strategy. We investigate two parameters: the number of features and the number of clusters, because both parameters impact performance and size of the working set. We investigate two settings for the number of clusters, $k = 4$ and $k = 64$. Note that N decreases for higher values of d, because we keep data size constant.

Observations on GPU. Cluster Merge and Partitioned Features show equal throughput if both parameters are set to small values, i.e., $k = 4$ and $d \leq 16$. However, Partitioned Features outperforms Cluster Merge when $k = 64$ or $d > 16$. The reason is that less work groups fit into local memory, because Cluster Merge's working set grows with higher values of k and d. Consequently, the GPU's hardware scheduler runs less work groups (i.e., lower *occupancy*), thus reducing parallelism. If $k \times d > 384$, Cluster Merge falls back to global memory, because the working set of a single work group does not fit into local memory. For Partitioned Features, we observe a small drop in performance for increasing k and d.

Observations on CPU. The throughput of Cluster Merge halves for both settings of k and continues to drop for $k = 64$. Partitioned Features is initially slower than Cluster Merge. However, Partitioned Features does not fall back to L2 cache for $k = 64$, because features are partitioned over multiple threads. Overall, Cluster Merge and Partitioned Features have similar performance on CPUs.

6.2.2 Mass Sum Strategies

In Figure 7, we evaluate the throughput of different Mass Sum strategies for a varying k. Since Mass Sum accesses only labels, d has no impact on performance.

Observations on GPU. Partitioned Private performs best until $k = 16$. After this point, the growing working set decreases the GPU occupancy, which decreases throughput. Starting from $k = 128$, Partitioned Local outperforms Partitioned Private. Partitioned Global's throughput increases above $k = 4$, as there are less write contentions. Global Atomic achieves very low throughput, due to write contention involving thousands of threads.

Observations on CPU. Partitioned Private has sufficient cache space for stable throughput. In contrast, Partitioned Local and Global achieve five times lower throughput than Partitioned Private. Global Atomic also has low throughput, despite having less write contentions than on GPU.

6.2.3 Runtime Performance

From the previous experiments, we observed that we achieve a fast centroid update using the Partitioned Feature strategy (Feature Sum) and the Partitioned Private strategy (Mass Sum). We derive three major strategies that we compare. The *Cross-Processing strategy* performs the point assign-

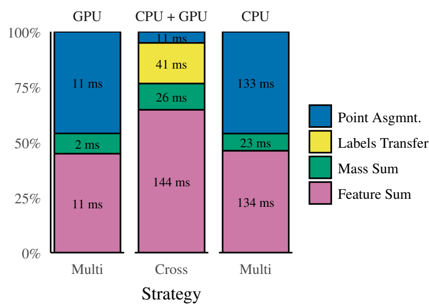


Fig. 9 Execution time breakdowns for Cross-Processing and Multi-Pass on CPU and GPU for 4 features and 4 clusters.

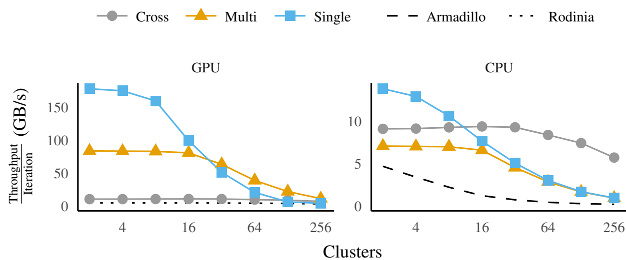


Fig. 10 Performance of strategies for varying number of clusters on CPU and GPU with 4 features.

ment on the GPU and performs the centroid update on the CPU, transferring the labels to the CPU between steps (state-of-the-art). The *Multi-Pass strategy* uses our fast centroid update and performs point assignment and centroid update on the same processor (Section 3). The *Single-Pass strategy* fuses our centroid update routine with point assignment and processes one k-means iteration with one pass over the point data (Section 4). In this experiment, we compare the overall strategy execution times per-processor and break down the individual execution times. In Figure 8, we show the execution times of all strategies for $k = 4$ and $d = 4$. For strategies with multiple phases, we show the relative time spent per phase in percent in Figure 9.

Observations. The Cross-Processing strategy is dominated by the Feature Sum and label transfers from GPU to CPU. On the CPU, the Multi-Pass strategy has the highest execution time and is dominated by point assignment and Feature Sum. The Single-Pass strategy halves the execution time because it needs to read the point data only once. On the GPU, the Multi-Pass strategy outperforms the Cross-Processing strategy by a factor of 9.1. The Single-Pass strategy improves the performance by a factor of 2 compared to the Multi-Pass strategy and by a factor of 19.3 compared to the Cross-Processing strategy.

6.2.4 Scaling Clusters

In Figure 10, we investigate the impact of different numbers of clusters k on the throughput. We observe that throughput generally decreases as k increases.

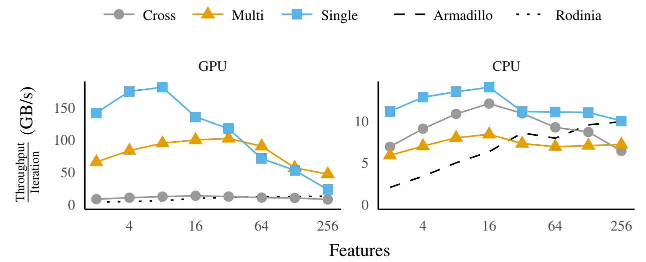


Fig. 11 Performance of strategies for varying number of features on CPU and GPU with 4 clusters.

Observations on GPU. The Single-Pass strategy outperforms the Multi-Pass strategy for $k \leq 16$ by up to a factor of 2. The Multi-Pass strategy outperforms the Single-Pass strategy starting from $k = 32$. Both strategies converge to the same performance of the Cross-Processing strategy starting from $k = 128$. This is because the point assignment dominates execution time with growing k.

Observations on CPU. The Single-Pass strategy outperforms the Cross-Processing strategy until $k \geq 16$. Starting from this point, the single and Multi-Pass strategies have very similar performance and are outperformed by the Cross-Processing strategy.[2]

6.2.5 Scaling Features

In Figure 11, we scale the number of features d and measure the throughput. We set the number of clusters to $k = 4$. As we keep data size constant with scaling d, the number of data points N halves with every doubling of d. Thus, computational complexity does not change for point assignment and Feature Sum. However, the computational complexity of Mass Sum scales with $\frac{1}{d}$.

Observations on GPU. The Single-Pass strategy outperforms the Multi-Pass strategy for $d < 32$. For greater d, the performance of both strategies is similar and outperforms Cross-Processing.

Observations on CPU. The Single-Pass strategy consistently outperforms the Multi-Pass strategy and the Armadillo baseline.

6.2.6 Chunk Transfers

We investigate the impact of chunk sizes on our chunk-wise transfer strategy in two parts: First, we observe only the transfer from main memory to GPU memory (see Figure 12(a)). Second, we observe a three-stage pipeline (see Figure 12(b)). In this pipeline, each chunk is copied into a pinned buffer in main memory, then transferred via PCI-e, and finally processed on the GPU. To measure the transfer, we call an empty GPU function on each chunk. We show the mean and standard deviation (if above 5%) over 100 trans-

[2] Note that the Cross-Processing strategy uses the GPU for point assignment, whereas Single-Pass and Multi-Pass are executed on CPU only. Therefore we include the Cross-Processing strategy in both plots.

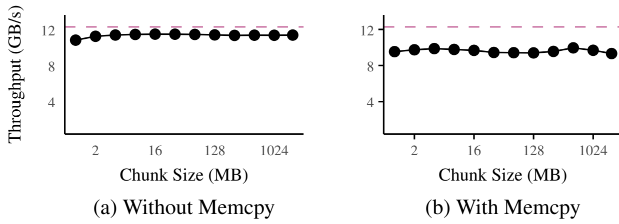


Fig. 12 Vary chunk sizes in transfer from main memory to GPU, performing (a) only transfer, and (b) memcpy-transfer-execute pipeline.

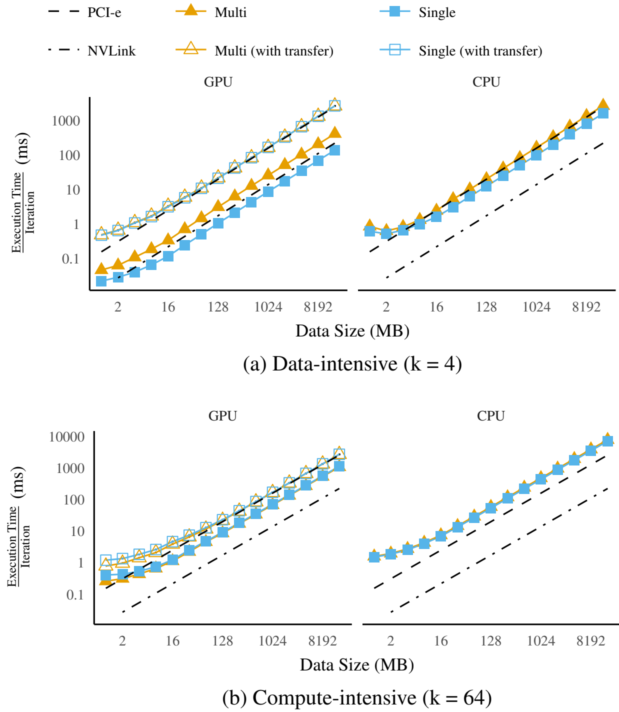


Fig. 13 Performance of strategies for increasing data size on CPU and GPU with (a) 4 and (b) 64 clusters, and 4 features.

fers. As an upper bound, we show the maximum bandwidth measured by Nvidia's bandwidth utility.

Observations. The transfer without memcpy() (up to 11.4 GB/s) nearly reaches the throughput limit of 12.2 GB/s. We achieve maximum throughput with chunk sizes between 8 and 64 MB. In contrast, when running the complete pipeline, we observe a maximum throughput of 9.8 GB/s with chunk sizes of 4, 8, 16, and 512 MB. However, all measurements are within 5% of the maximum observation.

Thus, we conclude that main memory copies slow down throughput by 20%. In contrast, chunk sizes have only a small impact on overall PCI-e throughput.

6.2.7 Data Scaling

In this experiment, we investigate the scalability of our Single-Pass and Multi-Pass strategies in the case when data exceeds the size of GPU memory (Figure 13). We scale the size of point data from 1 MB to 16 GB and consider data-intensive ($k = 4$) as well as compute-intensive ($k = 64$) scenarios with $d = 4$ features. We use 16 MB chunks, based on our results in Section 6.2.7. To emphasize the transfer process, we transfer all data to the GPU and do not cache any data in GPU memory.

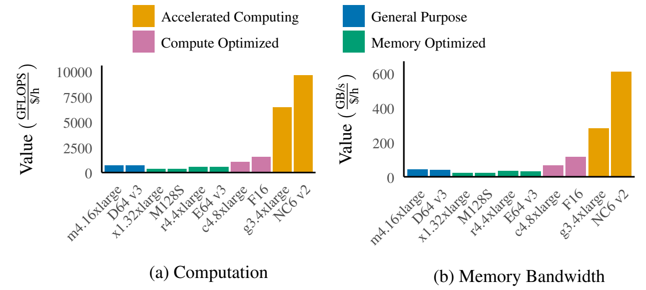


Fig. 14 Cost comparison of different EC2 instances for *k*-means.

Observations on GPU. Our Single-Pass and Multi-Pass strategies always cluster points with full PCI-e bandwidth. This is possible because we fully overlap data transfer with computation and our strategies complete the clustering before the next chunk arrives. We also show the pure execution times of both strategies and relate them to the bandwidths of PCI-e (measured) and the novel NVLink (projected) interconnects. In the data-intensive case, only the Single-Pass strategy is able to fully utilize the bandwidth of both PCI-e and NVLink interconnects. With NVLink, the Multi-Pass strategy would be bound by computation and not data transfer. In contrast, in the compute-intensive scenario both strategies perform equally, both strategies saturate PCI-e, and neither strategy would saturate NVLink.

Observations on CPU. We observe that the Single-Pass strategy consistently outperforms the Multi-Pass strategy in both scenarios. Furthermore, when compute-intensive, the GPU outperforms the CPU by a factor of up to 2.6. In the data-intensive scenario, the GPU performs as well as the CPU. The reason is that our CPU is incapable of clustering faster than the PCI-e bandwidth. Thus overall, the GPU processes data at least as fast as the CPU despite having to transfer data to the GPU. In line with related work [15, 40], in this experiment we show the feasibility of GPU coprocessing for data-intensive algorithms.

6.2.8 Comparing Performance per \$: CPU vs. GPU

In Figure 14, we compare the performance per dollar of different Amazon EC2 [2] instance models. We show theoretical peak performance for computation and memory bandwidth, with prices per hour. Amazon classifies instance models into specialized categories. We are primarily interest the GPU models classified as "Accelerated Computing", thus "g3.4xlarge" and "NC6 v2". All EC2 instances use recent Intel Haswell and Broadwell CPUs. "g3.4xlarge" instances are "r4.4xlarge" instances with an Nvidia Tesla M60. For GPU instances, we consider only GPU performance.

Observations. Out of all CPU instances, the "c4.8xlarge" model offers the best value for computation and for memory bandwidth, at $1.591 per hour for 1670.4 GFLOPS

and 60 GB/s (prices from 25. May 2018). Nevertheless, “g3.4xlarge” provides $6.1\times$ and $4.2\times$ better value in both metrics, at $1.140 per hour for 7365 GFLOPS and 320 GB/s. Compared to a base “r4.4xlarge”, the advantage is even $11.6\times$ and $8.7\times$, respectively, at $1.064 per hour for 588.8 GFLOPS and 34.16 GB/s.

We conclude that GPUs add at least $4\times$ the performance for each dollar to CPU instances. Thus, exploiting GPUs in EC2 represents a worthwhile investment for data analysis.

6.3 Discussion

Centroid Update Strategies. In our experiments, we showed that the Cluster Merge strategy of Feature Sum has a performance difference of two orders-of-magnitude between $d = 2$ and $d = 256$. In the worst case, Feature Sum is $4.6\times$ slower on GPU than on CPU, thus motivating the Cross-Processing strategy. In contrast, our Partitioned Features strategy improves such cases by up to $96.7\times$ on the GPU. Furthermore, we showed that, unlike on CPU, there is no single, best Mass Sum strategy on GPU. Rather, Partitioned Private performs up to $1.7\times$ faster than Partitioned Local while $k \leq 64$. Falling back from local to global memory incurs a penalty of $1.83\times$. In sum, Partitioned Features and Partitioned Private/Local lay the foundation for efficient centroid update on GPUs.

Run-Time Results. In analyzing *k*-means as a whole, we discovered that the Cross-Processing strategy (Centroid Update on CPU) is often more than ten times slower compared to the Multi-Pass strategy (centroid update on GPU). The main reason is the aforementioned slow centroid update in combination with the cross-processing problem. Furthermore, avoiding the multi-pass problem yields another factor of two between Multi-Pass and Single-Pass strategies on both processor types.

Scalability Results. We observed further, that parameters are impacted unequally. When scaling the number of clusters k, computational complexity increases, which transforms *k*-means from a memory-bound algorithm for $k \leq 32$ to a compute-bound algorithm for larger k. Even though computational complexity remains constant when scaling the number of features d for fixed k, performance decreases. If the memory footprint of point assignment grows, the GPU occupancy decreases and point assignment eventually falls back to global memory.

Performance per $. Finally, we showed that GPUs are a valuable investment. On Amazon EC2, a virtual instance with one GPU provides more than $4\times$ better compute and memory performance per dollar than instances without a GPU. Thus, for an equal amount of money, *k*-means can analyze more data and more diverse parameter values.

7 Related Work

While early GPU-based *k*-means implementations where limited to OpenGL [7, 37], recent languages such as OpenCL or CUDA are designed for computational tasks and enable advanced optimizations. In particular, these languages allow us to avoid materialization of intermediate results.

Avoiding materialization is especially relevant for assigning points to clusters, because it is space- and memory-intensive to materialize all point-to-centroid distances. This can be further optimized through caching centroids [13] and data points [24] in GPU local memory or L1 cache. Our point assignment subphase builds on this approach and adds low-level optimizations to further increase efficiency.

Optimizations that reduce the computational complexity of point assignment emphasize our work, because high k-values do not make *k*-means computation-bound. Specifically, Hall and Hart [16] apply Elkan’s kd-tree approach [11] to a GPU implementation. They propose to store centroids in a kd-tree, such that finding the nearest centroid requires less comparisons for large k. These optimizations are orthogonal and complementary to our work.

GPUMiner [12] reduces transfer overhead for the labels in the Cross-Processing strategy with bitmap compression. In contrast, we update centroids directly on the GPU with our Multi-Pass and Single-Pass strategies to eliminate this source of overhead entirely.

Although updating centroids on GPUs [24] or MICs [23] (i.e., Intel Xeon Phi) has been proposed previously, we evaluate this Multi-Pass strategy for a wide range of cluster and feature parameters. In particular, we show that Cluster Merge is inefficient on GPUs and introduce new optimizations with our cache-efficient Partitioned Features strategy. Regarding the individual subphases of centroid update, researchers proposed different solutions using GPUs. First, Feature Sum was implemented by using SQL group-by aggregation [20, 33], which has been implemented on CPUs [29] and GPUs [17, 21]. However, Feature Sum in *k*-means and relational-style group-by aggregation differ significantly. In particular, having tens or hundreds of features is common in data sets, but aggregating over this many attributes is uncommon in relational queries (e.g., TPC-H [1]). Thus, to the best of our knowledge, we are the first who optimize coprocessor-optimized grouped aggregation for this use case. Second, Mass Sum on GPU could exploit histogram computation [30]. These general implementations sort pixel values into buckets, which requires, e.g., divisions or branches. Frequent branch divergence reduces performance on GPUs [39]. In contrast, in our Mass Sum strategies, labels directly index into buckets and thus avoid branches. Furthermore, our Partitioned Local strategy uses hardware-native atomic operations in local memory, which were emulated in software at the time of previous work [32].

CPU implementations that compute *k*-means in a single data pass exist [28, 36]. However, on GPUs, we require a different approach because our Single-Pass strategy must reshuffle data between threads on-the-fly.

8 Conclusion

In this paper, we propose a GPU-optimized algorithm for *k*-means. Our algorithm centers around a highly-optimized strategy for updating centroids on GPUs. In our algorithm, we solve two fundamental problems of previous approaches: *cross-processing* and *multi-pass execution*. In contrast to previous approaches, we focus on reducing cache space usage through architectural features of GPU hardware, such that we are able to increase the effective parallelism. As a result, we propose a highly-optimized strategy for *k*-means that runs entirely on a GPU and requires only a single pass over the data. The evaluation shows that the Single-Pass strategy achieves up to $2\times$ and $20\times$ higher throughput than the Multi-Pass and Cross-Processing strategies, respectively. Finally, we show that our approach scales to large data sets exceeding the GPUs memory capacity. In our experiments, our GPU strategies performed at least as well as a CPU despite transferring data over PCI-e. Finally, our Single-Pass strategy was the only strategy that was capable of saturating the bandwidth of the NVLink interconnect for bandwidth-intensive scenarios.

Acknowledgments This work was funded by the EU projects SAGE (671500) and E2Data (780245), DFG Priority Program "Scalable Data Management for Future Hardware" (MA4662-5), and the German Ministry for Education and Research as BBDC (01IS14013A).

References

1. (2017) Transaction processing performance council. TPC-H. URL http://www.tpc.org/tpch
2. (2018) Amazon ec2 pricing. URL https://aws.amazon.com/ec2/pricing/on-demand
3. Arthur D, Vassilvitskii S (2007) k-means++: The advantages of careful seeding. In: ACM-SIAM, pp 1027–1035
4. Bai H, et al (2009) k-means on commodity GPUs with CUDA. In: WRI CSIE, pp 651–655
5. Breß S, Funke H, Teubner J (2016) Robust query processing in co-processor-accelerated databases. In: SIGMOD, pp 1891–1906
6. Breß S, et al (2017) Generating custom code for efficient query execution on heterogeneous processors. CoRR abs/1709.00700
7. Cao F, Tung AKH, Zhou A (2006) Scalable clustering using graphics processors. In: WAIM, pp 372–384
8. Cassou C (2008) Intraseasonal interaction between the madden–julian oscillation and the north atlantic oscillation. Nature 455(7212):523–527
9. Che S, et al (2009) Rodinia: A benchmark suite for heterogeneous computing. In: IISWC, pp 44–54
10. Dall M, et al (2017) Arctic sea ice melt leads to atmospheric new particle formation. Scientific reports 7(1):3318
11. Elkan C (2003) Using the triangle inequality to accelerate k-means. In: ICML, pp 147–153
12. Fang W, et al (2008) Parallel data mining on graphics processors. Tech. Rep. HKUST-CS08-07, HKUST
13. Farivar R, et al (2008) A parallel implementation of k-means clustering on GPUs. In: PDPTA, pp 340–345
14. Fernando R (2004) GPU gems: programming techniques, tips and tricks for real-time graphics, Pearson Higher Education, chap 37.2
15. Funke H, et al (2018) Pipelined query processing in coprocessor environments. In: SIGMOD, ACM
16. Hall J, Hart J (2004) GPU acceleration of iterative clustering. In: GPGPU, pp 45–52
17. He B, et al (2009) Relational query coprocessing on graphics processors. TODS 34(4)
18. Heimel M, et al (2013) Hardware-oblivious parallelism for in-memory column-stores. PVLDB 6(9):709–720
19. Heintzman ND, et al (2007) Distinct and predictive chromatin signatures of transcriptional promoters and enhancers in the human genome. Nature Genetics 39(3):311
20. Hellerstein J, et al (2012) The MADlib analytics library or MAD skills, the SQL. PVLDB 5(12):1700–1711
21. Karnagel T, Müller R, Lohman GM (2015) Optimizing GPU-accelerated group-by and aggregation. In: ADMS, pp 13–24
22. Kleisner KM, et al (2016) The effects of sub-regional climate velocity on the distribution and spatial extent of marine species assemblages. PLOS ONE 11:1–21
23. Lee S, et al (2016) Evaluation of k-means data clustering algorithm on intel xeon phi. In: BigData, pp 2251–2260
24. Li Y, et al (2010) Speeding up k-means algorithm by GPUs. In: IEEE CIT, pp 115–122
25. Lloyd S (1982) Least squares quantization in PCM. IEEE Trans Inf Theory 28(2):129–136
26. Lutz C, et al (2018) Efficient k-means on GPUs. In: DaMoN, DOI https://doi.org/10.1145/3211922.3211925
27. MacQueen J, et al (1967) Some methods for classification and analysis of multivariate observations. In: Proc. Fifth Berkeley Symp. on Math. Statist. and Prob., vol 1, pp 281–297
28. Mhembere D, et al (2017) knor: A NUMA-optimized in-memory, distributed and semi-external-memory k-means library. In: HPDC
29. Müller I, et al (2015) Cache-efficient aggregation: Hashing is sorting. In: SIGMOD, pp 1123–1136
30. Nugteren C, et al (2011) High performance predictable histogramming on GPUs: exploring and evaluating algorithm trade-offs. In: GPGPU, p 1
31. Nvidia (2017) CUDA C programming guide. Tech. Rep. PG-02829-001_v8.0, URL http://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf
32. Nvidia (2017) Tuning CUDA applications for maxwell. Tech. Rep. DA-07173-001_v9.0, URL http://docs.nvidia.com/cuda/pdf/Maxwell_Tuning_Guide.pdf
33. Passing L, et al (2017) SQL- and operator-centric data analytics in relational main-memory databases. In: EDBT, pp 84–95
34. Pirk H, Manegold S, Kersten ML (2014) Waste not... efficient co-processing of relational data. In: ICDE, pp 508–519
35. Pirk H, et al (2016) Voodoo - A vector algebra for portable database performance on modern hardware. PVLDB 9(14):1707–1718
36. Sanderson C, Curtin R (2016) Armadillo: a template-based c++ library for linear algebra. Journal of Open Source Software
37. Shalom A, Dash M, Tue M (2008) Efficient k-means clustering using accelerated graphics processors. In: DaWaK, pp 166–175
38. Shindler M, Wong A, Meyerson AW (2011) Fast and accurate k-means for large datasets. In: NIPS, pp 2375–2383
39. Sitaridi EA, Ross KA (2013) Optimizing select conditions on gpus. In: DaMoN, p 4
40. Stehle E, Jacobsen H (2017) A memory bandwidth-efficient hybrid radix sort on GPUs. In: SIGMOD, pp 417–432
41. Vitak SA, et al (2017) Sequencing thousands of single-cell genomes with combinatorial indexing. Nature Methods 14(3):302
42. Wu F, et al (2013) A vectorized k-means algorithm for Intel Many Integrated Core architecture. In: APPT, pp 277–294
43. Zang C, et al (2016) High-dimensional genomic data bias correction and data integration using mancie. Nature Communications 7:11305
44. Zhang T, Ramakrishnan R, Livny M (1996) Birch: An efficient data clustering method for very large databases. In: SIGMOD, pp 103–114