Don't Forget About Synchronization! A Case Study of K-Means on GPU

Jacob Nelson
Computer Science and Engineering
Lehigh University, USA
jjn217@lehigh.edu

Roberto Palmieri
Computer Science and Engineering
Lehigh University, USA
palmieri@lehigh.edu

Abstract

Heterogeneous devices are becoming necessary components of high performance computing infrastructures, and the graphics processing unit (GPU) plays an important role in this landscape. Given a problem, the established approach for exploiting the GPU is to design solutions that are parallel, without data or flow dependencies. These solutions are then offloaded to the GPU's massively parallel capability. This design principle (i.e., avoiding contention) often leads to developing applications that cannot maximize GPU hardware utilization. The goal of this paper is to challenge this common belief by empirically showing that allowing even simple forms of synchronization enables programmers to design parallel solutions that admit conflicts and achieve better utilization of hardware parallelism. Our experience shows that lock-based solutions to the k-means clustering problem outperform the well-engineered and parallel KMCUDA on both synthetic and real datasets; averaging 8.4x faster runtimes at high contention and 8.1x faster for low contention, with maximums of 25.4x and 74x, respectively. We summarize our findings by identifying two guidelines to help make concurrency effective when programming GPU applications.

CCS Concepts • **Software and its engineering → General programming languages**; • **Social and professional topics →** *History of programming languages*;

Keywords GPU, Synchronization, K-means, Concurrency

ACM Reference Format:
Jacob Nelson and Roberto Palmieri. 2019. Don't Forget About Synchronization! A Case Study of K-Means on GPU. In *The 10th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'19), February 17, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3303084.3309488

1 Introduction

Nowadays, general purpose multicore machines serve a vast class of different workloads, spanning from those produced by an individual user, to the ones targeted by market-leading companies. Although large in terms of parallelism, general purpose multicore processors cannot compete with the parallelism made available by dedicated heterogeneous devices. Consequently, exploiting these devices to increase performance (when possible) continues to increase in popularity.

By taking advantage of specialized hardware, tasks can run faster and more efficiently than by only using general purpose processors. For example, the graphics processing unit (GPU) has become an important tool for a variety of problems, including machine learning, simulations, and other computationally expensive applications whose execution can be parallelized. Its popularity is attributed to its generous number of cores (e.g., 2560 for the NVIDIA GTX-1080 and 5120 for the NVIDIA Titan V).

Unfortunately, heterogeneity often comes at the cost of decreased programmability. Many efforts have already been made to simplify the programmer's life by transparently bridging different instruction set architectures (e.g., [5, 7, 8, 34]). Normally, optimizations enabled by the device's native programming language are the first candidates not to be ported. Another important consequence of heterogeneity is the need to re-engineer algorithms to meet hardware requirements and ultimately achieve the desired speedup over the original (CPU-based[1]) code. We focus on this particular aspect of GPU-based computing, tackling a common belief that significant speedup cannot be reached without re-engineering.

Generally, certain portions of any algorithm that are inherently parallel represent good candidates to be offloaded to the GPU, if its contribution to runtime is noticeable compared to the overall application runtime [6, 22]. For sections of a program where synchronization might be needed, some solutions prefer to use GPU `atomics` (e.g., [1]), where others choose to pass execution to the CPU, where synchronization is simpler to address [25]. Recent research on GPU synchronization provides more advanced solutions. Some examples of this are, transactional memory [9, 11, 12, 19, 36], locking algorithms [35], scheduling algorithms [18] and even architectural changes [17].

We propose that even with elementary constructs, synchronization should be considered viable when writing GPU code. We support our claims by describing a case study on the widely used k-means clustering algorithm (originally designed to help digitally represent analog signals [28] and used extensively for a wide variety of clustering tasks [13–15, 23, 26, 33]). In the evaluation of our lock-based implementations for the GPU, we discover that handling concurrency with fine-grain spinlocks can lead to more than 20x average speedup against a carefully crafted and well-engineered parallel version (i.e., KMCUDA [3]). Take note that our motivation is to champion the use of fine-grain synchronization in GPUs, not to design a high-performance variant of the k-means algorithm.

Our investigation is also orthogonal to work done to improve synchronization on the GPU. For example, transactional memory has been proposed (both in software and hardware) [9, 11, 12, 19, 36]

PMAM'19 , February 17, 2019, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6290-0/19/02…$15.00
https://doi.org/10.1145/3303084.3309488

[1]When we reference CPU programming, we are referring specifically to non-heterogeneous x86-compliant code.

Algorithm	Centroid Memory Location	Lock Granularity
GM-CL	Global	Per-cluster
GM-DL	Global	Per-dimension
SM-CL	Shared	Per-cluster
SM-DL	Shared	Per-dimension

Table 1. Overview of the four lock-based algorithms.

to coordinate accesses to common data, as well as advanced lock-based synchronization [24, 35] techniques. In this paper we do not aim at presenting new strategies for implementing synchronization on GPUs; rather, we address the situation of taking an existing problem and designing a solution that leverages synchronization before re-engineering the algorithm to be completely parallel. However, improving concurrency controls will facilitate the adoption of concurrent GPU programming.

2 Overview

Despite the considerable amount of work produced on synchronization on the GPU, it is a commonly held belief that fine-grain synchronization on the GPU should be avoided whenever possible [2]. In essence, when programming for the GPU it is considered better to redesign an existing algorithm to be completely parallel (i.e., no data dependency among parallel executing tasks) rather than attempting to gain speedup with code that may cause data races. The decision between re-engineering an application to take advantage of the GPU parallelism and adopting an existing solution for CPU is often driven by the effort needed to carry out the re-engineering, which might not be predictable if the problem to be solved is custom.

Fine-grain synchronization also enables certain algorithms, which may better exploit the massively parallel hardware offered by the GPU. Ignoring fine-grain synchronization automatically eliminates approaches that might outperform carefully parallelized solutions. In this paper we do not argue that shared-nothing parallelism ought to be avoided, but that other options should also be considered when designing high performance GPU programs. Our goal is not to answer when synchronization should be preferred over a parallel design, but it empirically demonstrates that synchronization can provide higher performance, as is seen in our case study. Generalizations of its usefulness will be a future direction.

Many approaches to synchronization on the GPU target architectural changes [9, 17, 19] to facilitate implementing mutual exclusion and manufacturers are responding accordingly. For instance, NVIDIA's Volta architecture now supports independent thread scheduling by maintaining a program counter and stack for each thread [31]. This enhances capabilities for fine grain synchronization by eliminating difficulties arising from single-instruction, multiple threads (SIMT) execution.

2.1 Methodology

In order to achieve our goal of assessing performance of traditional forms of synchronization (e.g., locks) in the context of GPUs, we design four lock-based implementations of the k-means update phase with carefully chosen characteristics to understand how the GPU architecture affects performance. We focus on two aspects: lock granularity and shared memory.

To clarify the design decisions made in each of the four algorithms, we briefly summarize how k-means organizes data and processes it (more details in Section 4). K-means is a clustering algorithm; where a *centroid* (or cluster center) is the average of all members of a cluster and maintains the same number of dimensions as a data point. Because conflicts are possible when calculating centroids, each cluster must be protected by a lock, which is stored as metadata. Section 5 goes into more detail, but the following is an overview of the four algorithms we propose (an overview of the configurations is given by Table 1):

- *Global-Memory Centroid-Lock* (GM-CL) stores all centroids and metadata in global memory (accessible by all threads) and each cluster center is protected by a single lock.
- *Global-Memory Dimension-Lock* (GM-DL) makes lock granularity finer by protecting each dimension of a cluster center with a lock. Like GM-CL, all centroids and metadata are stored in global memory.
- *Shared-Memory Centroid-Lock* (SM-CL) protects each centroid with a single lock, but stores centroids and their metadata in shared memory (i.e., a faster on-chip memory that restricts access to a subset of threads).
- *Shared-Memory Dimension-Lock* (SM-DL) utilizes shared memory but maintains locks for every dimension of a cluster.

We compare our algorithms against the competitor, KMCUDA, by running our experiments on a NVIDIA GTX-1080 GPU. The parameters for each experiment are the number of data points to be clustered, n, the dimensionality of data points and clusters, d, and the number of clusters to find, k. For the following results, we set $d = 32$ and k varies. Two datasets are used. The first is synthetic data drawn from a uniform distribution. Admittedly not naturally clustered, this avoids biases introduced by pre-clustered data and represents the worst case scenario. Additionally, we test the Corel Image Feature Dataset to verify that our findings apply to real-world data.

For the synthetic dataset, GM-CL achieves an average speedup of 1.9x across all k for $n = 5000$ and 4.4x for $n = 50000$ while observing maximum speedups of 4.2x and 13.5x respectively. GM-DL reaches average speedups of 6.1x for $n = 5000$ and 21x for $n = 50000$ with maximums of 14x and 75x. We observe average speedups of 5.2x for $n = 5000$ and 7.2x for $n = 50000$ when running SM-CL. This implementation attains a maximum speedup of 12x for $n = 5000$ and 25x for $n = 50000$. SM-DL only observes a maximum speedup of 2.2x for $n = 5000$ and 2.6x for $n = 50000$, with both average speedups across all values of k falling below 1x.

For the real-world dataset, GM-CL achieves an average 3x improvement over the competitor and maximum of 12.9x when $k = 4096$. GM-DL averages 9.4x with a max of 37.4x, again at $k = 4096$. SM-CL averages 9.6x with a maximum of 22.9x at $k = 256$. SM-DL only averages 2.3x, but its max of 4.9x occurs at $k = 2$.

As a general result, increasing granularity decreases the work necessary for each thread, but it can also saturate the GPUs many cores. Using shared memory is faster, but can lead to unnecessary overhead when the cost of managing data overpowers the decreased latency of using shared memory.

2.2 Contributions

- We implement four lock-based solutions that solve k-means using the GPU.

- Contrary to common belief, we demonstrate that GPU lock-based programming is a viable (and in our experience superior) alternative to re-engineering an algorithm to avoid data races.
- We provide two general guidelines for designing lock-based algorithms on the GPU based on the experimental findings.

The rest of the paper is organized as follows: First, Section 3 describes fundamental concepts of GPU computing as they pertain to our work. Next, Section 4 describes the k-means algorithm and previous efforts to write GPU implementations. Section 5 outlines our lock-based k-means algorithms. Section 6 evaluates our experimental results and Section 7 gives general guidelines for lock-based GPU implementations. Finally, Section 8 concludes the paper.

3 GPU Background

GPUs are designed to compute massively parallel workloads efficiently. Execution is based on single instruction multiple threads, or *SIMT*, processing. In both execution and memory, GPU hardware organization enforces properties that directly influence performance. For this reason, the rest of this section overviews these concepts.

Programming Model. GPUs allow thousands of threads to process in parallel. A *kernel* determines the behavior of many threads cooperating to do work. The *host* (i.e., the CPU) launches a kernel on the *device* (i.e., the GPU), which then executes the kernel. The total number of threads executing a kernel is predetermined by the program before it is launched. Once executing, access to thread ID metadata allows decisions to be made on a per-thread basis, for example indexing into an array of cluster centers.

Execution Model. A kernel is composed of a grid of *thread blocks*, or groups of threads that share hardware. In current architectures, each thread block is assigned to a *streaming multiprocessor* (SM) and continues executing on the same SM for its lifetime. A single SM has many processing units (e.g., 128 cores for the NVIDIA GTX-1080). One or more thread blocks can be scheduled to the same SM, but this number is restricted by maximum number of resident threads allowed by the hardware.

During execution, a subset of sequentially numbered threads from a block, known as *warp*, is scheduled and run by the SM. Due to SIMT processing, threads in a warp execute in lockstep [27], meaning that all threads execute the same instruction. In the case that two threads in a warp diverge due to a branch, an *active threads mask* determines which threads make progress. The SM disables non-active threads until the current branch is completed, after which it executes all divergent branches in the same manner described. After all divergent branches complete, the warp converges, which entails it can continue executing as a whole.

Memory Organization. For our purposes we restrict the discussion of memory on GPUs to global and shared memory. Other memories exist (e.g., texture memory and surface memory) but our work does not address them. They are specialized memories mostly used for graphics processing.

- *Global memory* is functionally equivalent to main memory of CPU. An L2 cache is accessible to all SMs and is serviced differently depending on if the data is also cached in the L1 cache. Typically, the L1 cache is reserved for read-only data, however compiler directives provide functionality to force caching all data in L1.
- *Shared memory* is on chip memory that is local to each SM, and therefore only shared by threads in a thread block. The amount of shared memory allocated per thread block is fixed prior to kernel launch.

A common pattern is to copy data into shared memory, process it using the faster memory, then copy it back to global memory [21]; providing an essential optimization for many GPU applications.

SIMT Mutual Exclusion. Dealing with data races on GPUs is challenging due to the lockstep execution of threads, memory organization, and poor programmability. While modern GPUs offer atomic operations (e.g., `atomicAdd()`, `atomicSub()`, `atomicCAS()`), they lack more expressive APIs (e.g., `pthread_mutex_locks`), which constitute the fundamental building blocks in designing and developing correct concurrent programs on CPU.

The major pitfall in implementing concurrency abstractions, such as lock, is that threads diverge when a lock is already taken leading to sequential execution of divergent branches. In addition, some instrumentation is required to guarantee the progress of a warp, namely making sure all the divergent paths will eventually make progress.

A (frequent) deadlock condition occurs when threads in a warp diverge. Specifically, when a divergent thread holds the lock needed by a non-divergent thread. When a thread spins until a lock is acquired (e.g., by using "`while(CAS!=1)`"; in C++), it causes threads that successfully acquired locks to become divergent while the non-divergent threads execute the empty body of the while loop. When this happens, threads that successfully acquired a lock are masked off and will not execute the critical section until the non-divergent threads exit the while loop. When a lock needed by a non-divergent (i.e., spinning) thread is held by a divergent thread, then the non-divergent thread will spin forever and branches can never reconverge. Deadlock occurs because a divergent thread cannot make progress until a non-divergent thread finishes, which will spin on the while loop indefinitely because it requires a lock a divergent thread holds.

A common solution [4] maintains a flag denoting if a thread has completed its critical section and replacing the while condition with this flag. A thread will then try to acquire the lock inside the while loop. If it fails, the thread will simply pass over the critical path and wait for divergent branches (i.e., threads that were able to get the lock) before entering the while loop again. This allows successful threads to finish the critical section and release their locks. All threads in a warp will then check the while condition again, repeating the above process.

Additionally, due to the absence of powerful APIs to handle concurrency, it is the programmers responsibility to handle anomalies due to weak memory consistency models. Alglave et al. in [4] use carefully constructed programs to force weak memory behaviors on the GPU. They show that, to correct this behavior, fences and volatile memory are needed to ensure valid memory access. This does not differ from implementing a lock for the CPU. In fact, we leverage our knowledge of concurrent CPU programming to implement our lock-based k-means algorithms for the GPU.

4 K-means Background

K-means is a clustering algorithm for unsupervised learning and is commonly used [13–15, 23, 26, 33]. Given a data set (of n number of points each with d dimensions), k clusters are found that best represent the underlying groups of data. Portions of k-means are easily parallelized, while others can cause conflicts if not carefully designed. This section overviews solutions for solving k-means suited for both CPU and GPU. We selected implementations that are simple, as well as implementations tailored for better performance.

4.1 K-means Algorithms: Lloyd & Yin-Yang

The classic k-means algorithm, Lloyd's algorithm [28], is straightforward to understand. It consists of two phases. First, after initializing k centroids (by assigning them to random points in the data set, for example), the assignment phase assigns each data point to the closest cluster based on some distance metric between the point and the cluster's center. Then, the update phase uses the data points assigned to a cluster to set the cluster centroid to the average of all its members. It repeats the assignment and update phases until all clusters' members do not change or some number of iterations exceeds a fixed threshold.

Another solution to k-means, Yin-Yang (YY) [16], is considered a drop in replacement for Lloyd's k-means and has been shown to improve performance on CPU by up to 10x, compared to Lloyd's algorithm (see comparison in [16]).

YY optimizes the assignment phase by using global and group filters to reduce the number of necessary comparisons, resulting in a large speedup. In addition to reducing the amount of work done during this phase, YY approaches the update phase differently, as well. Instead of computing the average of all members, it makes slight alterations to a centroid based on changes in the assignment of a data point. Updates are only made using data points whose assignments change to or from the cluster. This allows for easy parallelization of the computation because each thread can scan the entire data set for points whose membership changes relative to themselves and do the update accordingly, without interfering with other clusters. Another benefit is that when the number of reassignments in an iteration are low, few calculations are needed.

4.2 Implementations of K-means for GPU

As previously stated, k-means consists of two computationally intensive phases: the assignment and update phases. There is also an initialization phase, which precedes the assignment and update phase but only happens once during the entire computation.

In both Lloyd and YY, the assignment phase is inherently parallel because writes do not generate conflicts. However, the update phase is where potential data races are possible. For both algorithms, this can be avoided by parallelizing over the centroids, but this can lead to under-utilization when compared to parallelizing over the data points because in k-means the number of data points generally surpasses the number of clusters to find.

In the following paragraphs we discuss solutions to implement k-means for GPUs. Each approaches the update phase differently to handle (or eliminate) conflicts.

CUDA k-means. This algorithm is provided as a library by NVIDIA [10]. Instead of using a lock-based approach, it formulates k-means to rely entirely on atomic additions, since write conflicts occur when data points are members of the same centroid.

KMCUDA. KMCUDA uses the Yin-Yang k-means algorithm. Because the assignment phase does not introduce possible data races, we will not include a description of how KMCUDA achieves the assignment phase on the GPU. For more information on the implementation details, both their code [3] and an accompanying write-up [29] are available.

To avoid synchronization in the update phase, KMCUDA assigns each thread to update a single cluster. As discussed, each thread updates its designated cluster based on how assignments change. It relies on shared memory by storing the assignments for each data point for faster memory access. First, centroids are scaled by the number of members assigned to them in the previous iteration. Next, an array in shared memory is filled with both current and previous assignments for as many data points as can fit in shared memory. Threads then iterate through the assignments. If the point was previously in a thread's cluster but now is not, then the data point is subtracted from the scaled centroid. If the data point now belongs to the thread's cluster, the point is added to the total. In any other case, the point is skipped.

Remaining data points are processed by repeatedly filling shared memory with the next batch of assignments and updating the centroids accordingly.

Finally, the resulting intermediate value is divided by the new count to find the new centroid. During execution, both centroids and data set are kept in global memory. KMCUDA's strategy allows performance to scale well with respect to the number of clusters because each thread is assigned a centroid to update. Additionally, performance scales reasonably well with respect to the number of data points. This occurs because global memory accesses are minimized to only those points for which the point's assignment changes with respect to the thread's cluster.

Speeding up K-Means Algorithm by GPU. An approach proposed by Zhao et. al in [25] computes temporary centroids on the GPU and then copies them back to the CPU for final processing. During the subsequent iteration, these new centroids must be copied back to the GPU. This data transfer incurs a large penalty, which dominates the update time. Although this pattern can be translated to other algorithms, the cost of transferring data between the host and device has a major impact in the overall application runtime.

5 Design

To support our claim that offloading concurrent algorithms to the GPU should be considered as a feasible and less laborious alternative to providing a parallel (i.e., without data races) solution of a given problem, we design four lock-based algorithms implementing Lloyd's k-means. These algorithms draw from well-known strategies for concurrent CPU programming while targeting optimizations specific to GPUs.

For our implementation, we split the update phase into two kernels: a summation kernel and a normalization kernel. The summation kernel calculates a sum of all members for each cluster and also records the number of members belonging to each cluster. The normalization kernel then divides each sum by the cluster's total number of members to find the new cluster center. The latter is trivial to implement and does not introduce any conflicts, however the former contains data dependency. To understand the effects of our lock-based synchronization on the GPU, our implementations differ exclusively in the summation portion of the update phase.

Similar to CPUs, weak memory behaviors exist on the GPU [4]. For this reason, it is necessary to declare all communally accessed and updated data as volatile and use memory fences to guarantee visibility of concurrent updates on shared memory locations for all threads. In contrast, optimizations unique to the GPU (i.e., massive parallelism and shared memory) are an integral part of developing high performance GPU applications. We leverage both a traditional concurrency approach and GPU specific architecture to realize lock-based mutual exclusion on GPU.

K-means can either be parallelized over the data points or the clusters; each strategy has their advantages. Recall from Section 4 that in order to eliminate conflicts, parallel implementations must parallelize over clusters. Each thread is responsible for updating one cluster. In this way, there are never two threads attempting to update the same cluster. However, this is not ideal given that typically the number of clusters to be found with k-means are far fewer than the size of the data set [20]. By parallelizing the problem over the data set instead of the clusters, a concurrent implementation has potentially greater speedup over a non-conflicting version.

In lieu of looping through the entire data set, a thread is responsible for processing only a single data point. The thread adds the point to the cluster it belongs to, and increments that cluster's membership counter. As a consequence, conflicting updates can occur because many data points may be members of the same cluster. To avoid conflicts, we use a spinlock, which mimics a CPU implementation, to synchronize memory accesses between threads. Handling concurrent access via locks eliminates the need to completely re-engineer the problem to fit the GPU by drawing on existing knowledge from concurrent CPU programming paradigms.

When handling concurrency, we aim to target two attributes specific to GPU architecture. First, thousands of threads are available on GPUs, which is far greater than even large multicore CPUs. Harnessing this characteristic for our lock-based algorithms, threads can either be responsible for a single data point or a single dimension of a data point. For the latter, we also use finer-grain locks. Additionally, algorithm data and metadata (i.e., locks and membership count) may reside in either global or shared memory. As described in Section 3, shared memory is faster than global memory. Decreased memory latency allows threads to both acquire/release locks and update centroids faster. Furthermore, due to shared memory being private to each thread block, only inter-thread-block contention is possible, inherently limiting potential conflicts to only the threads running within a block (more details in Section 5.3).

In the following subsections, we describe the details of each of our algorithms. Each algorithm was verified on a small dataset with known cluster centers to assess correctness.

5.1 Global Memory Cluster-Lock (GM-CL)

GM-CL allocates one lock per cluster and stores all cluster data and metadata in global memory. During the critical path, it serially updates each dimension of a centroid. One positive consideration of this algorithm is that it is easy to implement because it does not leverage shared memory.

Each thread is assigned a single data point to process. Based on the cluster the point belongs to, the thread attempts to acquire the lock protecting the cluster. If successful, it performs an element wise addition of the data point to the centroid and increments the cluster's membership counter (stored in the algorithm's metadata). When finished, it releases the lock for use by other threads. When a thread is unsuccessful, its branch diverges causing the thread to wait for all successful threads within its warp to complete before reattempting lock acquisition. This is an artifact of the lock-step execution of SIMT processing. Figure

After all the summation is complete, the normalization kernel divides each cluster centroid by the associated members counter to find the updated center.

5.2 Global Memory Dimension-Lock (GM-DL)

One simple way to improve upon GM-CL is to reduce each thread's workload and use finer-grain locks. Specifically for GM-DL, each thread is responsible for only one dimension of a data point and only updates the same dimension of the corresponding cluster center. Additionally, each dimension of a cluster is protected by a lock, increasing the lock granularity compared to GM-CL.

Apart from the previously mentioned differences with respect to GM-CL, the algorithm follows a similar behavior. A thread acquires the lock for a given centroid dimension, adds the point's corresponding dimension to the running total, and increments the membership counter, only if it is responsible for the first dimension. Once complete, it releases the lock. Finally, the normalization kernel runs to divide each centroid by the number of members for each cluster.

5.3 Shared Memory Cluster-Lock (SM-CL)

SM-CL exploits shared memory. Because shared memory is limited, there is a possibility that not all cluster data and metadata (i.e., locks) can fit at one time. If the memory needed for all clusters exceeds what is available on the architecture, the algorithm updates the cluster centers in chunks.

As is the case in all of our lock-based implementations, before launching the kernel requests enough threadblocks such that every thread considers at most one data point. For each iteration, after the clusters are initialized in shared memory, threads update centroids based on the thread's assigned data point. Threads whose data point is a member of a cluster in the current chunk in shared memory make progress for that iteration, while the others wait their turn.

As mentioned, data in shared memory are only visible to threads within the same thread block, therefore, after the clusters are updated locally, one thread per thread block is assigned to update the versions in global memory. The centroids reside in global memory, maintain locks and are updated similarly to GM-CL.

One advantageous side effect of using shared memory is the capability of managing the contention among parallel threads. In fact, because shared memory can only be seen by threads in the same thread block, and because the number of running threads in the same thread block is restricted to the number of cores on an SM, the maximum possible contention on clusters in shared memory is reduced when compared to contention for clusters in global memory, which all threads can access.

Because shared memory is private to each thread block, the theoretical maximum contention on clusters in shared memory is equal to the number of threads in a thread block. However, based on our current architecture and configuration, we ensure that only one thread block will be scheduled per SM. A thread block will run on the same SM for its lifetime; therefore the possible contention on each lock residing in shared memory is bound by the number of cores per streaming multiprocessor.

5.4 Shared Memory Dimension-Lock (SM-DL)

SM-DL merges both fine granularity and shared memory. SM-DL differs from SM-CL in the following ways. First, the amount of metadata to be stored increases by a factor of the number of dimensions, since each dimension must keep a lock. Additionally, a thread's critical path is reduced to only processing a single dimension of a data point. Although finer granularity increases the level of parallelism, there is more metadata to handle, which consumes space in shared memory at the cost of storing fewer clusters in shared memory. Accordingly, to process the same amount of data, the algorithm must execute more iterations to calculate all clusters.

5.5 Shared-Nothing Parallel Algorithms

We also implement four parallel algorithms that exploit the same granularity and memory mechanisms (i.e., global and shared memory) as the lock-based approaches but avoid synchronization by parallelizing over clusters. For these algorithms, metadata only consists of the membership count. The first, named GM-CT (Global Memory, Cluster per Thread), assigns a cluster to each thread, storing data and metadata in global memory. The thread loops through all data points and adds members to the cluster's running sum, incrementing the membership count along the way.

In the second approach, GM-DT (Global Memory, Dimension per Thread), a thread is responsible for only one dimension of a cluster. Again, each thread loops through all data points, updates the corresponding dimension of members of the thread's cluster, and increments the membership count.

The two remaining algorithms (SM-CT and SM-DT) use the same approaches as GM-CT and GM-DT, but store all data and metadata in shared memory.

As will be clear later, these algorithms are unable to perform well because they do not allow the same degree of parallelism.

6 Evaluation

To demonstrate the viability of fine-grain synchronous programming on the GPU, we compare our four different lock-based implementations of k-means, namely GM-CL, GM-DL, SM-CL, SM-DL, against our competitor, KMCUDA. As discussed in Section 5, each algorithm varies in lock granularity and memory location (i.e., global or shared). Each run consists of the size of the data set, n, the dimensionality of data, d, and the number of clusters used by k-means, k.

All tests are executed on an Intel Core i7-7700k processor with a NVIDIA GeForce GTX-1080 GPU. Code is written in C++ and GPU kernels in CUDA 9.1 for NVIDIA compute capability 6.1 (Pascal). The GPU used in our experiments has 20 streaming multiprocessor units (SMs), 128 cores per SM, 8 GB of main memory and up to 49KB of shared memory per thread block. For all runs, thread blocks consist of 1024 threads. This is the maximum number available.

We begin our study by using the synthetic dataset and analyzing how algorithms behave at different values of k, which acts as a knob to control the level of contention in the system. When generating data sets, values for each dimension are taken from a uniform distribution from 0 to 1. Again, using a uniformly distributed dataset removes potential bias from the data making comparisons more fair. Later in Figure 6, we show that the trends seen in the synthetic dataset hold in a real-world scenario. To compare algorithms, we record the average time of the update phase for each algorithm. It is important to note that this measurement includes the normalization kernel because in KMCUDA it is implicitly done during centroid update. Initially, we test each implementation on a data set of 5000 points with 32-dimensional data. Next, we increase n to 50000 to understand behavior for larger data sets. Finally, we return to $n = 5000$ but vary data set dimensionality, d, for three distinct values of k.

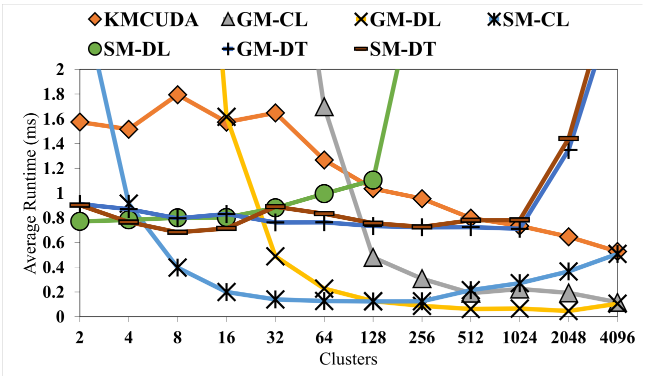


Figure 1. Average run time of the update phase for each algorithm for $n = 5000$. The lower the run time the better.

As a summary of our findings, the best performing lock-based strategies are GM-DL and SM-CL, depending on the level of contention within the system. When contention is high, SM-CL alleviates conflicts by managing the possible contention on clusters and reduces memory access latency. However, when contention is low, GM-DL is faster because threads are less likely to conflict and the critical path is shorter, both due to finer lock granularity. Two of our purely parallel implementations (i.e., GM-DT and SM-DT) outperform KMCUDA only when all requested thread blocks can be simultaneously scheduled to SMs. The other two parallel algorithms, GM-CT and SM-CT, never offer benefits over the competitor and are therefore ignored to increase readability.

Going into the details of our evaluation, we first analyze the behavior of our competitor KMCUDA. Recall from Section 4 that while the Yin-Yang algorithm employed in KMCUDA is generally fast due to reducing the number of necessary comparisons during the assignment phase, it also optimizes the update phase by computing new centroids in parallel without conflicts. Specifically, each centroid is only updated when data points change membership with respect to its cluster. Based on this and the fact that one thread processes each centroid, average runtimes for KMCUDA (shown in Figure 1 depend on the number of reassignments made. If there are few reassignments, then threads do little work. A consequence of this is that average runtimes for KMCUDA are directly affected by how close initial centroids are to the optimal solution. The variability in the lower values of k reflects this dependency.

Our lock-based approaches demonstrate more predictable behavior, determined mainly by contention on the shared clusters and by resource restrictions (e.g., available threads).

The simplest strategy, GM-CL, performs very poorly when contention is high because of waiting threads. When more clusters are used, conflicting memory accesses are less likely to occur. Figure 1 shows that after $k = 64$ lock contention no longer dominates the runtime. In fact, from $k = 64$ to $k = 4096$, this algorithm has an average speedup of 3.1x over KMCUDA. The maximum speedup occurs at $k = 512$ with 4.2x boost in performance (speedups over KMCUDA reported in Figure 2).

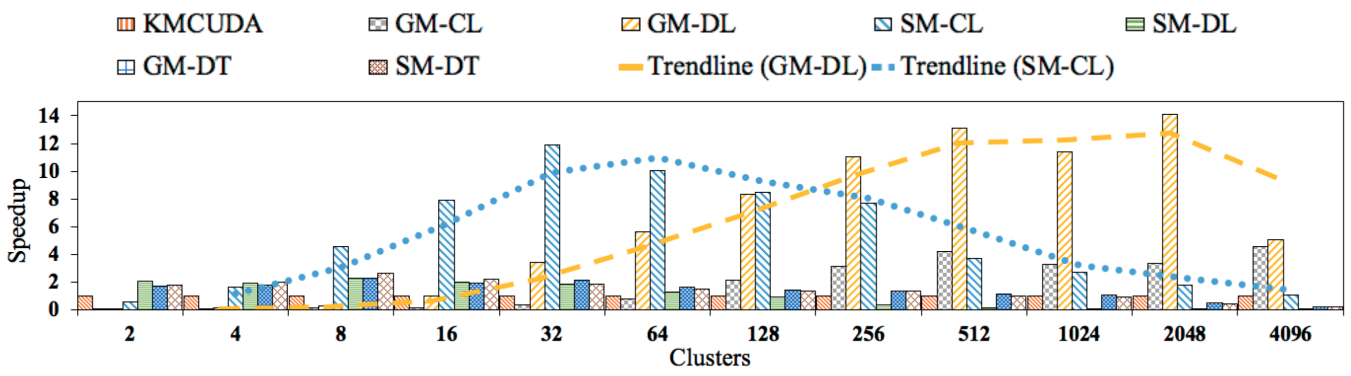


Figure 2. Average speedup over KMCUDA of the update phase for each algorithm for $n = 5000$. The higher the speedup the better. The trendlines show the moving average for both GM-DL and SM-CL.

Next we evaluate GM-DL, the finer-grained version of GM-CL. On average, between $k = 64$ and $k = 4096$ GM-DL offers 8.1x improvement over KMCUDA. This approach is able to produce a maximum speedup of 14.1x when $k = 2048$. Generally, this algorithm outperforms KMCUDA, with the exception that for $k < 32$ the algorithm suffers from performance hits due to high lock contention. Generally, the trend of GM-DL is similar to that of GM-CL but with increased benefits. This is due to finer lock granularity, allowing more threads to make progress, as well as a shorter critical path because each thread is only responsible for a single dimension. These characteristics help GM-DL to perform well when processing many clusters.

In both global memory implementations, the primary performance bottleneck, aside from lock contention, is the number of available hardware threads.

After considering different granularity, we introduce shared memory. In general and as expected, shared memory helps when the overhead of managing it does not dominate the runtime. In k-means, shared memory is most beneficial when a single lock protects each cluster (i.e., SM-CL). To recall, SM-CL first initializes cluster placeholders in shared memory, updates them according to the data points assigned to each thread, then updates the corresponding versions stored in global memory.

Contrary to GM-CL, SM-CL performs well at smaller values of k. SM-CL provides an average speedup over KMCUDA of 8.4x from $k = 8$ to $k = 256$. Its maximum speedup is 12x, when $k = 32$. Notably, because shared memory is used, contention in shared memory is constrained to the number of running threads and contention on global data is limited by the number of SMs. The former is unique to GPU hardware organization and processing constraints. Remember that shared memory is private to each thread block, meaning only threads in the same block may access it. Not only is contention limited by block size, but also, because thread blocks cannot span two SMs, contention is limited to the number of processing units available on one SM. Contention on cluster in global memory is reduced because only one thread per thread block is assigned to update the centroids in global memory, using the versions computed in shared memory.

Shared memory only has a capacity of 49KB per thread block, which becomes saturated at $k = 256$. Beyond that point, not all clusters and metadata can fit in shared memory, which decreases the performance benefits (see Figure 1).

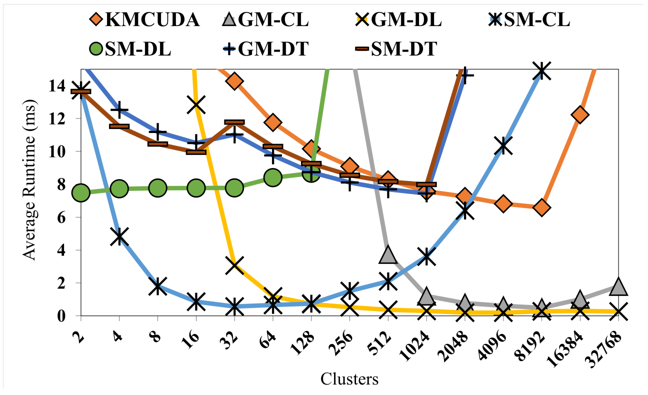


Figure 3. Average runtimes of the update phase of k-mean for each algorithm with $n = 50000$.

SM-DL also offers improvements over KMCUDA, but only for $2 < k < 64$. At $k = 128$, shared memory can no longer hold all data and metadata so multiple iterations must be performed to process all clusters. In this case, the benefits of shared memory are dominated by the cost of managing that memory, which results in poor performance.

We conclude our experiments for $n = 5000$ by addressing the two parallel algorithms, GM-DT and SM-DT. GM-DT stores all cluster data and metadata in global memory, while SM-DT stores them in shared memory. For both algorithms, each thread processes a dimension of a cluster. We observe speedups only when all thread blocks can fit on the GPU (i.e., when $k < 2048$). For larger values of k, we see a linear increase in run time due to the inability to simultaneously schedule all required thread blocks.

To understand how our lock-based approaches scale with respect to the number of data points, we perform the same tests discussed above with $n = 50000$. In general, the previously described trends hold for the larger data set as can be seen in Figure 3 (runtime) and Figure 4 (speedup). Again, GM-DL performs best when contention is low, while SM-CL performs best when contention is high. Similarly, the two plotted parallel versions, GM-DT and SM-DT, require more thread blocks than what can be simultaneously scheduled, leading to poor performance when $k > 1024$.

One upshot of using more data points is that it gives an understanding of algorithm behavior when hardware resources become

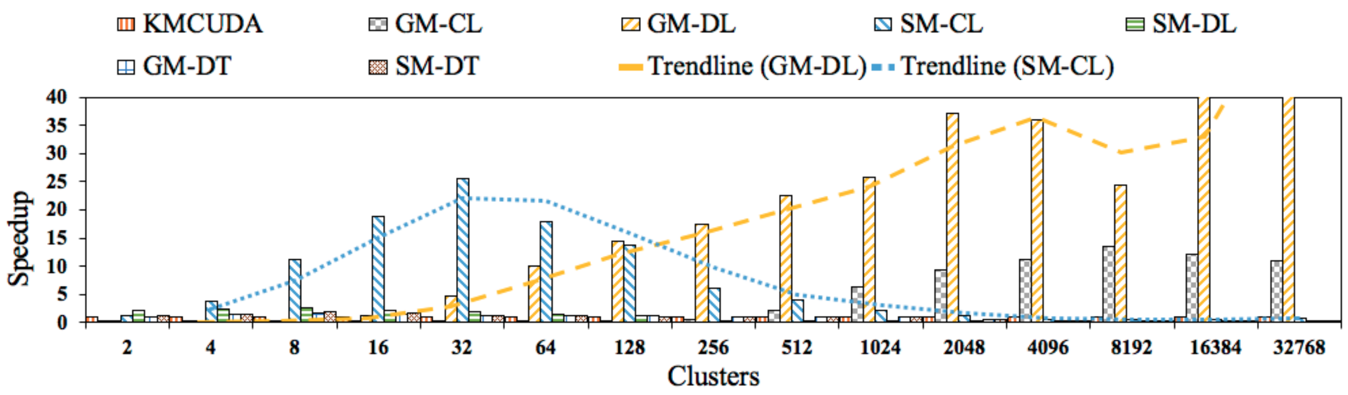
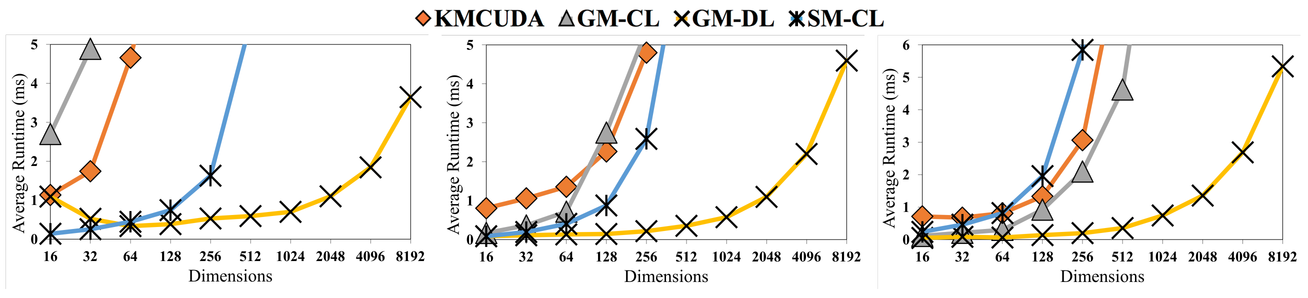


Figure 4. Speedup of each algorithm over KMCUDA for varied values of k. For GM-DL and SM-CL we include trendlines for the moving averages of each implementation.



(a) k = 32 - High Contention **(b)** k = 256 - Moderate Contention **(c)** k = 2048 - Low Contention

Figure 5. Three plots demonstrating how the best performing algorithms behave when varying the dimensionality of the data set and the number of clusters, k. For all of them, the data set consists of 5000 points.

the primary constraint. For example, all data can fit in the L2 cache for n = 5000 but this is not the case for higher values of n. Although the overall shape of KMCUDA's runtime is the same for both configurations of n. However, the runtime for n = 50000 is more than 10x slower than n = 5000 due to cache line eviction. Additionally, with larger values of k, KMCUDA requires more shared memory than is available. Particularly when $k > 8192$, the previous and current assignments can no longer fit in shared memory causing the average update time to increase suddenly.

In contrast to our competitor, GM-CL and GM-DL continue to operate as before; they perform well at large numbers of clusters and perform poorly for low k. Having more data points, compared to n = 5000, increases the possibility of conflicting memory accesses. As a result, runtimes for GM-CL are more than 10x slower than for n = 5000 for $k < 1024$. GM-DL takes advantage of fine-grain locking to achieve faster runtimes earlier, producing an average speedup over KMCUDA of 20.7x across all k and a maximum speedup of 74.8x at k = 32768.

As is the case for n = 5000, the two shared memory algorithms (SM-CL and SM-DL) perform better than their global memory counterparts at lower k values, but eventually become restricted by the size of shared memory. Exactly as happens to KMCUDA, these algorithms can no longer store the required cluster data and metadata in shared memory, forcing them to iterate more than once. For $k > 128$, the cost of shared memory management outweighs its benefits. In the case of SM-DL, surpassing this threshold results in a sudden increase of the average update time as seen in Figure 3. SM-CL requires fewer total iterations to pass all clusters through shared memory, because more clusters can fit per iteration, thus the increase in runtime is less dramatic. SM-CL averages 7.2x speedup across all k and a maximum speedup of 25.5x when k = 32. Importantly, for $k < 1024$ the average speedup is 11.3x and 0.97x for $k \geq 1024$, demonstrating that shared memory is good when contention is high.

We end the discussion by commenting on the two relevant parallel implementations, GM-DT and SM-DT. Again, we do not include the versions which process one cluster per thread because in our experiments they never perform better than any other implementation. The sudden increase in average runtime for both algorithms (GM-DT and SM-DT) occurring at k = 1024 results from the hardware restriction on simultaneously running thread blocks. In fact, our testbed makes 20 SMs available, with two thread blocks running on each for a total of 40 concurrently running thread blocks. For k greater than 1024 clusters, the algorithm allocates more than 40 thread blocks causing some thread block to wait until an SM is available before being scheduled to run.

In addition to testing different values of n, we also run experiments to understand the implications of data set dimensionality, d. Figure 5 shows the results. All tests are run with a data set consisting of 5000 data points, sampled from a uniform distribution between 0 and 1. We test KMCUDA, GM-CL, GM-DL and SM-CL.

SM-DL is not included because it cannot fit all of the necessary data and metadata in shared memory for large d.

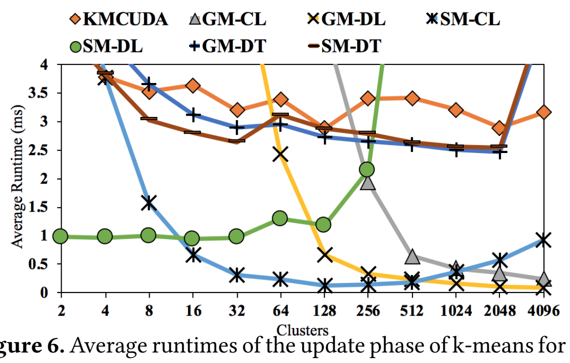


Figure 6. Average runtimes of the update phase of k-means for each algorithm for edge data from the real-world Corel Image Feature Data Set.

Figure 5a shows the average update times as d ranges from 16 to 8192 for $k = 32$. This value of k represents settings where SM-CL is fastest. For low dimensional data ($k < 64$), SM-CL is the fastest to update clusters because it takes full advantage of shared memory. However, GM-DL becomes the better algorithm at larger values of d. Finer granularity and shorter critical path give it an advantage over the other algorithms in terms of runtime.

We also run the same experiment for $k = 256$ (see Figure 5b). With this test we are targeting moderate contention in the system. As for $k = 32$, the results demonstrate that increasing d reduces the effectiveness of SM-CL. Fewer clusters are able to fit in shared memory causing more iterations to occur. Although the plot is cropped to increase readability, SM-CL eventually becomes slower than GM-CL for $k > 1024$.

Finally, in Figure 5c we configure $k = 2048$. Here, contention is lowest between the three values of k and GM-DL remains the obvious winner for all values of d.

To understand the implications of using synthetic data sets, we also test a real-world data set to validate that the observed results are not a simply a function of the data itself. For this we use a subset of the Corel Image Features Data Set [32] provided by the Stanford Transactional Applications for Multi-processing (STAMP) project [30]. The data set consists of 17695 images represented by two features: color and edge. The color data has 9 dimensions, while the edge data consists of 18 dimensions. In the interest of space, we only report the results for the edge data but note that the observed trends also appear in the color data. Figure 6 displays the average runtime for each algorithm as a function of k. The primary difference between the real world data set and the contrived data is the behavior of KMCUDA, which does not improve with higher values of k for a real-world data set. These results support our findings and show worse performance for our competitor as k increases, thus showing that the trends we found are orthogonal to the particular data used.

7 Discussion

In this section, we overview our findings specific to k-means and our evaluation study, then, we generalize those findings to concurrent programming on the GPU.

7.1 K-Means Findings

As a final remark of our evaluation study, we clearly identify two implementations (GM-DL and SM-CL) that leverage fine-grain synchronization to provide at least competitive, and most of the time significantly better, performance than the implementation optimized to be parallel without code dependency. The lock-based algorithms we design limited GPU specialized work required by the programmer, while giving speedup over the competitor in most scenarios.

As previously observed, when values of k are low and lock contention is high, SM-CL performs well. This is not only because of faster memory accesses, but also because the hardware inherently limits the number of threads that may conflict at a given memory address. Recall that the GPU used in our experiments has 128 cores per SM. Because shared memory is private to each thread block and each thread block runs on one SM, only 128 threads may conflict on shared memory locations. In contrast, global memory implementations have potentially thousands of threads attempting to access the same memory. In addition to restricting the potential contention, only one thread per thread block executes when updating clusters in global memory. Again, in this case the hardware restricts contention because there is a limited number simultaneously running thread blocks (e.g., 20 for our configurations on the GTX-1080).

While shared memory is beneficial when contention is high, it is also limited to low dimensional data sets. When the memory required to store all clusters and their metadata surpasses the shared memory available, global memory becomes the attractive option. Indeed, GM-CL and GM-DL update clusters faster than the shared memory versions for high dimensional data sets. In the end, GM-DL provides the best generalized performance. Furthermore, it is the shortest (24 lines of code) implementation of the update phase; compared to SM-DL (66 lines) and KMCUDA (71 lines).

7.2 General Guidelines

After designing and implementing a number of lock-based algorithms we evaluated how well they compared to KMCUDA, a highly engineered parallel solution to clustering. Our results demonstrate that re-engineering is not necessarily the best solution, but that using a basic spinlock implementation can lead to good results.

Two common patterns emerge from our evaluation (shown by the moving average trendlines in Figure 2 and Figure 3). First, when contention is high, algorithms can benefit from storing the data in shared memory. Specifically, we saw that SM-CL has the fastest average update time when contention is higher. Second, finer lock granularity led to better performance due to a limited critical path and lower contention (as seen by GM-DL).

Our results culminate in the following guidelines based on our experiences implementing lock-based k-means.

(*G1*) When contention is high, prefer shared memory.
(*G2*) Fine grain synchronization complements the GPUs massively parallel architecture.

Regarding G1, capitalizing on the speed of shared memory gives the best performance for high contention situations. Although resource restrictions can counteract its benefit, shared memory also forces a hierarchical solution. Thread blocks compute locally (i.e., in shared memory) then push their results to global memory. A layered strategy considerably reduces conflicts in both shared and global memory.

Regarding G2, when expected contention is moderate to low, the best option is to target the characteristic unique to GPUs, namely their massively parallel architecture. Both finer division of work and finer grained locking give notable performance benefits.

In general, by following the proposed guidelines, G1 and G2, programmers have a competitive alternative to develop solutions for problems that otherwise would require a redesign in order to eliminate data dependency and execute parallel on GPU. Allowing concurrency enables better GPU computing resource utilization because it relaxes constraints needed to support parallelism where conflicts are avoided explicitly by design.

8 Conclusion

In this paper we implemented four lock-based k-means solutions for the GPU to demonstrate that handling concurrency (even in the form of naive spinlocks) can be beneficial on the GPU, and might save the effort of re-engineering existing solutions to eliminate data races. Furthermore, we provided guidelines for GPU lock-based algorithms based on our findings. Specifically, our results suggested that shared memory can benefit concurrent applications when contention is high, and that fine granularity is helpful to take advantage of the parallelism available on GPUs. Intentionally ignoring fine-grain synchronization on the GPU can lead to missed opportunities to fully utilize its capabilities. Future work will focus on extending the experiments to evaluate multi-GPU systems and designing flexible APIs that can dynamically elect a locking scheme based on available resources and runtime information.

Acknowledgments

This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0367.

References

[1] 2015. GPU Pro Tip: Fast Histograms Using Shared Atomics on Maxwell. https://devblogs.nvidia.com/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/
[2] 2017. Try to use lock and unlock in CUDA. https://devtalk.nvidia.com/default/topic/1014009/try-to-use-lock-and-unlock-in-cuda/
[3] 2018. KMCUDA. https://github.com/src-d/kmcuda
[4] Jade Alglave, Mark Batty, Alastair F Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU concurrency: Weak behaviours and programming assumptions. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 577–591.
[5] AMD. 2016. ROCm, a New Era in Open GPU Computing. https://rocm.github.io
[6] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units.* ACM, 94–103.
[7] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: bridging the programmability gap in heterogeneous-ISA platforms. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, Laurent Réveillère, Tim Harris, and Maurice Herlihy (Eds.). ACM, 29:1–29:16.
[8] Andrew Baumann, Paul Barham, Pierre-Évariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 29–44.
[9] A. Brownsword, W. W. Fung, I. Singh, and T. M. Aamodt. 2012. Kilo TM: Hardware Transactional Memory for GPU Architectures. *IEEE Micro* 32 (03 2012), 7–16.
[10] Bryan Catanzaro and Levi Barnes. 2015. NVIDIA K-means. https://github.com/NVIDIA/kmeans
[11] Daniel Cederman, Philippas Tsigas, and Muhammad Tayyab Chaudhry. [n. d.]. Towards a Software Transactional Memory for Graphics Processors.
[12] S. Chen, L. Peng, and S. Irving. 2017. Accelerating GPU hardware transactional memory with snapshot isolation. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA).* 282–294.
[13] Adam Coates and Andrew Y Ng. 2012. Learning feature representations with k-means. In *Neural networks: Tricks of the trade.* Springer, 561–580.
[14] Gabriella Csurka, Christopher Dance, Lixin Fan, Jutta Willamowski, and Cédric Bray. [n. d.]. Visual categorization with bags of keypoints.
[15] Inderjit S Dhillon and Dharmendra S Modha. 2001. Concept decompositions for large sparse text data using clustering. *Machine learning* 42, 1-2 (2001), 143–175.
[16] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In *International Conference on Machine Learning.* 579–587.
[17] A. ElTantawy and T. M. Aamodt. 2016. MIMD synchronization on SIMT architectures. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 1–14.
[18] A. ElTantawy and T. M. Aamodt. 2018. Warp Scheduling for Fine-Grained Synchronization. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 375–388.
[19] Wilson WL Fung and Tor M Aamodt. 2013. Energy efficient GPU transactional memory via space-time optimizations. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 408–420.
[20] Greg Hamerly and Charles Elkan. 2004. Learning the k in k-means. In *Advances in neural information processing systems.* 281–288.
[21] Mark Harris. 2013. Using Shared Memory in CUDA C/C++. https://devblogs.nvidia.com/using-shared-memory-cuda-cc/
[22] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 204–216.
[23] S. C. Lai and P. Y. Lau. 2018. Upper body action classification for multiview images using K-means. In *2018 International Workshop on Advanced Image Technology (IWAIT).* 1–4.
[24] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. 2015. Fine-Grained Synchronizations and Dataflow Programming on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15).* ACM, New York, NY, USA, 109–118.
[25] Y. Li, K. Zhao, X. Chu, and J. Liu. 2010. Speeding up K-Means Algorithm by GPUs. In *2010 10th IEEE International Conference on Computer and Information Technology.* 115–122.
[26] Dekang Lin and Xiaoyun Wu. 2009. Phrase clustering for discriminative learning. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2-Volume 2.* Association for Computational Linguistics, 1030–1038.
[27] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (March 2008), 39–55.
[28] S. Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (March 1982), 129–137.
[29] Vadim Markovtsev. 2016. Towards Yinyang K-means on GPU. https://blog.sourced.tech/post/towards_kmeans_on_gpu/
[30] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on.* 35–46.
[31] NVIDIA. 2018. Compute Capability 7.x. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability-7-x
[32] Michael Ortega-Binderberger, Kriengkrai Porkaew, and Sharad Mehrotra. 1999. Corel Image Features Data Set. data retrieved from University of California Irvine Machine Learning Repository, http://archive.ics.uci.edu/ml/datasets/Corel+Image+Features.
[33] O. J. Oyelade, O. O. Oladipupo, and I. C. Obagbuwa. 2010. Application of k Means Clustering algorithm for prediction of Students Academic Performance. *CoRR* abs/1002.2425 (2010). arXiv:1002.2425
[34] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test* 12, 3 (May 2010), 66–73.
[35] Yunlong Xu, Lan Gao, Rui Wang, Zhongzhi Luan, Weiguo Wu, and Depei Qian. 2016. Lock-based Synchronization for GPU Architectures. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '16).* ACM, New York, NY, USA, 205–213.
[36] Y. Xu, R. Wang, N. Goswami, T. Li, and D. Qian. 2014. Software Transactional Memory for GPU Architectures. *IEEE Computer Architecture Letters* 13, 1 (Jan 2014), 49–52.