# Optimizing OpenCL Kernels for Iterative Statistical Applications on GPUs

Thilina Gunarathne
Indiana University,
Bloomington, IN 47405, USA
tgunarat@cs.indiana.edu

Bimalee Salpitikorala
Indiana University,
Bloomington, IN 47405, USA
ssalpiti@cs.indiana.edu

Arun Chauhan
Indiana University,
Bloomington, IN 47405, USA
achauhan@cs.indiana.edu

Geoffrey Fox
Indiana University,
Bloomington, IN 47405, USA
gcf@cs.indiana.edu

## ABSTRACT

We present a study of three important kernels that occur frequently in iterative statistical applications: K-Means, Multi-Dimensional Scaling (MDS), and PageRank. We implemented each kernel using OpenCL and evaluated their performance on an NVIDIA Tesla GPGPU card. By examining the underlying algorithms and empirically measuring the performance of various components of the kernel we explored the optimization of these kernels by three techniques: 1. selectively placing data in different memory levels, 2. rearranging data in memory. and 3. dividing the work between the GPU and the CPU. The optimizations resulted in performance improvements of up to 5X, compared to naïve OpenCL implementations. We believe that these categories of optimizations are also applicable to other similar kernels. Finally, we draw several lessons that would be useful in not only implementing other similar kernels with OpenCL, but also in devising code generation strategies in compilers that target GPGPUs through OpenCL.

## 1. INTRODUCTION

Iterative algorithms are at the core of the vast majority of scientific applications, which have traditionally been parallelized and optimized for large multi-processors, either based on shared memory or clusters of interconnected nodes. As GPUs have gained popularity for scientific applications, computational kernels used in those applications need to be performance-tuned for GPUs in order to utilize the hardware as effectively as possible.

Often, when iterative scientific applications are parallelized they are naturally expressed in a bulk synchronous parallel (BSP) style, where local computation steps alternate with collective communication steps [26]. An important class of such iterative applications are statistical applications that process large amounts of data. A crucial aspect of large data processing applications is that they can often be fruitfully run in large-scale distributed computing environments, such as clouds.

In this paper, we study three algorithms, which we refer to as *kernels*, that find use in such iterative statistical applications. The intended environment to run these applications is loosely-connected and distributed, which could be leveraged using a cloud computing framework, such as MapReduce. In this paper, we focus on characterizing and optimizing the kernel performance on a single GPU node. The three kernels are:

1. K-Means, which is a clustering algorithm used in many machine learning applications;
2. MDS, which is a set of statistical techniques to visualize higher dimensional data in three dimensions; and
3. PageRank, which is an iterative link analysis algorithm relying on sparse matrix-vector multiplication.

These kernels are characterized by high ratio of memory accesses to floating point operations, thus necessitating careful latency hiding and memory hierarchy optimizations to achieve high performance. We conducted our study in the context of OpenCL, which would let us extend our results across hardware platforms. We studied each kernel for its potential for optimization by:

1. Utilizing OpenCL *local memory*, by software-controlled caching of selected data;
2. Reorganizing data in memory, to encourage hardware-driven memory access coalescing or to avoid bank conflicts; and
3. Dividing the computation between CPUs and GPUs, to establish a software pipeline across iterations.

We present detailed experimental evaluation for each kernel by varying different algorithmic parameters. Finally, we draw some lessons linking algorithm characteristics to the optimizations that are most likely to result in performance improvements. This has important implications not only for kernel developers, but also for compiler developers who wish to leverage GPUs within a higher level language by compiling it to OpenCL.

## 2. BACKGROUND

Boosted by the growing demand for gaming power, the traditional fixed function graphics pipeline of GPUs have

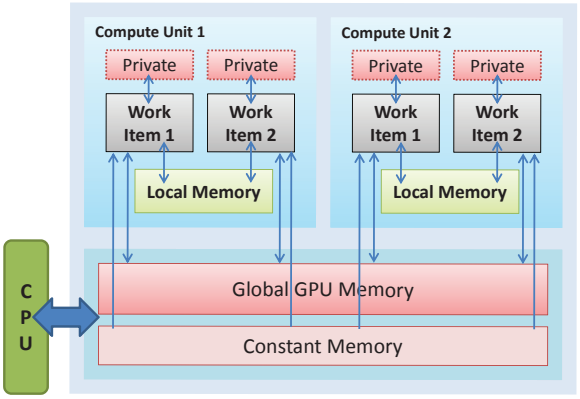



**Figure 1: OpenCL memory hierarchy. In the current NVIDIA OpenCL implementation, private memory is physically located in global memory.**

evolved into a full-fledged programmable hardware chain [14].

In this paper we use NVIDIA Tesla C1060 GPGPU card for our experiments. Tesla C1060 consists of 240 processor cores and 4 GB global memory with 102 GB/sec peak memory bandwidth. It has a theoretical peak performance of 933 GFLOPS for single precision and 78 GFLOPS for double precision.

It is the general purpose relatively higher level programming interfaces, such as OpenCL, that have paved the way for leveraging GPUs for general purpose computing. OpenCL is a cross-platform, vendor-neutral, open programming standard that supports parallel programming in heterogeneous computational environments, including multi-core CPUs and GPUs [10]. It provides efficient parallel programming capabilities on both data parallel and task parallel architectures.

A *compute kernel* is the basic execution unit in OpenCL. Kernels are queued up for execution and OpenCL API provides a set of events to handle the queued up kernels. The data parallel execution of a kernel is defined by a multi-dimensional domain and each individual execution unit of the domain is referred to as a *work item*, which may be grouped together into several *work-groups*, executing in parallel. Work items in a group can communicate with each other and synchronize execution. The task parallel compute kernels are executed as single work items.

OpenCL defines a multi level memory model with four memory spaces: private, local, constant. and global as depicted in Figure 1. Private memory can only be used by single compute units, while global memory can be used by all the compute units on the device. Local memory (called *shared memory* in CUDA) is accessible in all the work items in a work group. Constant memory may be used by all the compute units to store read-only data.

## 3. ITERATIVE STATISTICAL APPS

Many important scientific applications and algorithms can be implemented as iterative computation and communication steps, where computations inside an iteration are independent and are synchronized at the end of each iteration through reduce and communication steps. Often, each iteration is also amenable to parallelization. Many statistical applications fall in this category. Examples include clustering algorithms, data mining applications, machine learning algorithms, data visualization algorithms, and most of the expectation maximization algorithms. The growth of such iterative statistical applications, in importance and number, is driven partly by the need to process massive amounts of data, for which scientists rely on clustering, mining, and dimension-reduction to interpret the data. Emergence of computational fields, such as bioinformatics, and machine learning, have also contributed to an increased interest in this class of applications.

Advanced frameworks, such as Twister [9], can support optimized execution of iterative MapReduce applications, making them well-suited to support iterative applications in a large scale distributed environment, such as clouds. Within such frameworks, GPGPUs can be utilized for execution of single steps or single computational components. This gives the applications the best of both worlds by utilizing the GPGPU computing power and supporting large amounts of data. One goal of our current study is to evaluate the feasibility of GPGPUs for this class of applications and to determine the potential of combining GPGPU computing together with distributed cloud-computing frameworks. Some cloud-computing providers, such as Amazon EC2, are already moving to provide GPGPU resources for their users. Frameworks that combine GPGPU computing with the distributed cloud programming would be good candidates for implementing such environments.

Two main types of data can be identified in these statistical iterative applications, the loop-invariant input data and the loop-variant delta values. Most of the time, the loop-invariant input data, which remains unchanged across the iterations, are orders of magnitude larger than the loop-variant delta values. These loop-invariant data can be partitioned to process independently by different worker threads. These loop-invariant data can be copied from CPU memory to GPU global memory at the beginning of the computation and can be reused from the GPU global memory across iterations, giving significant advantages in terms of the CPU to GPU data transfer cost. To this end, we restrict ourselves to scenarios where the loop-invariant computational data fit within the GPU memory, which are likely to be the common case in large-scale distributed execution environments consisting of a large number of GPU nodes. Loop-variant delta values typically capture the result of a single iteration and will be used in processing of the next iteration by all the threads, hence necessitating a broadcast type operation of loop-variant delta values to all the worker threads at the beginning of each iteration. Currently global memory is used as the broadcast medium. We plan on exploring other broadcast mechanisms, such as constant memory, in the future.

It is possible to use software pipelining for exploiting parallelism across iterations. Assuming that only one kernel can execute on the GPU at one time, Figure 2 shows a scheme for exploiting loop-level parallelism. This assumes that there are no dependencies across iterations. However, if the loop-carried dependence pattern is dynamic, i.e., it may or may not exist based on specific iterations or input data, then it is still possible to use a software pipelining approach to *speculatively* execute subsequent iterations concurrently and quashing the results if the dependencies are detected. Clearly, this sacrifices some parallel efficiency. Another scenario where such pipelining may be useful is when the loop-carried dependence is caused by a convergence test.

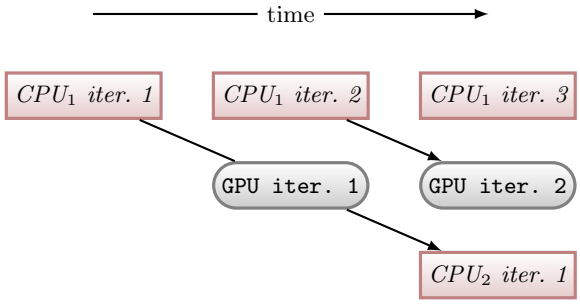



**Figure 2: Software pipelining to leverage GPUs for loop-level parallelism.**

In such a case, software pipelining would end up executing portions of iterations that were not going to be executed in the original program. However, that would have no impact on the converged result.

Note that if multiple kernels can be executed concurrently and efficiently on the GPU then the pipelining can be replicated to leverage that capability.

A characteristic feature of data processing iterative statistical applications is their high ratio of memory accesses to floating point operations, making them memory-bound. As a result, achieving high performance, measured in GFLOPS, is challenging. However, software-controlled memory hierarchy and the relatively high memory bandwidth of GPGPUs also offer an opportunity to optimize such applications. In the rest of the paper, we describe and study the optimization on GPUs of three representative kernels that are heavily used in iterative statistical applications.

# 4. K-MEANS CLUSTERING

Clustering is the process of partitioning a given data set into disjoint clusters. Use of clustering and other data mining techniques to interpret very large data sets has become increasingly popular with petabytes of data becoming commonplace. Each partitioned cluster includes a set of data points that are *similar* by some clustering metric and differ from the set of data points in another cluster. K-Means clustering algorithm has been widely used in many scientific as well as industrial application areas due to its simplicity and the applicability to large data sets [20].

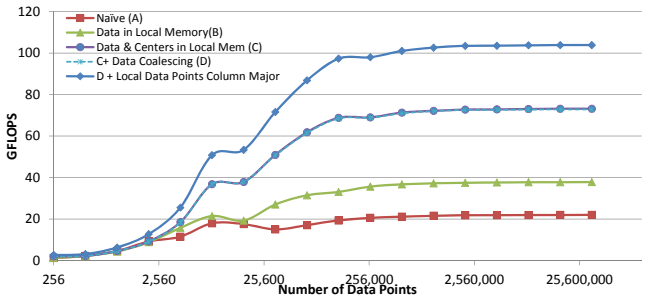



**Figure 3: K-Mean performance with the different optimizations steps, using 2D data points and 300 centroids.**

K-Means clustering algorithm works by defining $k$ *centroids*, i.e., cluster means, one for each cluster, and associating the data points to the nearest centroid. It is often implemented using an iterative refinement technique, where each iteration performs two main steps:

1. In the cluster *assignment step*, each data point is assigned to the nearest centroid. The distance to the centroid is often calculated as Euclidean distance.
2. In the *update step*, new cluster centroids are calculated based on the data points assigned to the clusters in the previous step.

At the end of iteration $n$, the new centroids are compared with the centroids in iteration $n-1$. The algorithm iterates until the difference, called the *error*, falls below a predetermined threshold. Figure 4 shows an outline of our OpenCL implementation of the K-Means algorithm.

The number of floating-point operations, $F$, in OpenCL K-Means per iteration per thread is given by $F = (3DM + M)$, resulting in a total of $F*N*I$ floating-point operations per calculation, where $I$ is the number of iterations, $N$ is the number of data points, $M$ is the number of centers, and $D$ is the dimensionality of the data points.

Figure 3 summarizes the performance of our K-Means implementation using OpenCL, showing successive improvements with optimizations. We describe these optimizations in detail in the remainder of this section.

```
__kernel KMeans(__global matrix,
  __global centroids,__global assignment,
  __local localPoints,__local localData){

 gid = get_global_id(0);
 lid = get_local_id(0);
 lz = get_local_size(0);

// Copying centroids to shared memory
 if (lid < centersHeight){
   for (int i=0; i < WIDTH ; i++){
    localPoints[(lid*WIDTH)+i] =
       centroids[(lid*WIDTH)+i];
   }
 }

// Copying data points to shared memory
 for (int i=0; i < WIDTH ; i++){
   localData[lid+(lz*i)] =
                matrix[(gid)+(i* height)];
 }
 for (int j = 0; j < centersHeight; j++){
   for (int i = 0; i < width; i++){
     distance = (localPoints[(j*width)+i]
                 - localData[lid +(lz*i)]);
     euDistance += distance * distance;
   }
   if (j == 0) {min = euDistance;}
   else if (euDistance < min) {
     min = euDistance; minCentroid = j;
   }
 }
 assignment[gid]=minCentroid;
}
```

**Figure 4: Outline of K-Means in OpenCL.**

## 4.1 Leveraging Local Memory

In the naïve implementation, both the centroid values as

well as the data points are accessed directly from the GPU global memory, resulting in a global memory read for each data and centroid data point access. With this approach, we were able to achieve performance in the range of 20 GFLOPs and speedups in the range of 13 compared to single core CPU[1].

The distance from a data point to each cluster centroid gets calculated in the assignment step of K-Means, resulting in reuse of the data point many times within a single thread. This observation motivated us to modify the kernel to copy the data points belonging to a local work group to the local memory, at the beginning of the computation. This resulted in approximately 75% performance increase over the naïve implementation, as the next line, marked "B", shows.

Each thread iterates through the centroids to calculate the distance to the data point assigned to that particular thread. This results in several accesses (equal to the local work group size) to each centroid per local work group. To avoid that, we copied the centroid point to the local memory before the computation. Caching of centroids values in local memory resulted in about 160% further performance increase, illustrated in the line marked "C" in Figure 3.

## 4.2 Optimizing Memory Access

As the next step, we stored the multi-dimensional data points in column-major format in global memory to take advantage of the hardware coalescing of memory accesses. However, this did not result in any measurable performance improvement as the completely overlapped lines "C" and "D" show, in Figure 3.

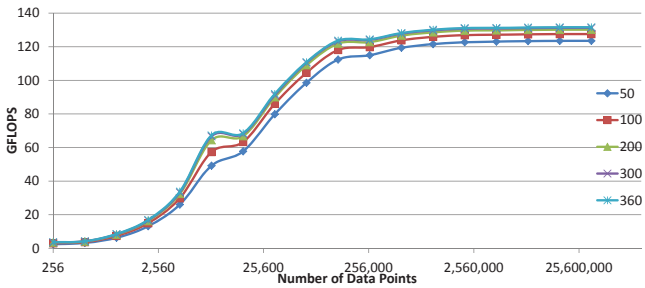
However, storing the data points in local memory in column-major format resulted in about 140% performance improvement, relative to the naïve implementation, represented by the line marked "D + shared data points . . . " in Figure 3. We believe that this is because of reduced bank conflicts when accessing local memory concurrently by different threads in a local work group. Performing the same transformation for centroids in local memory did not result in any significant change to the performance (not shown in the figure). We believe this is due to all the threads in a local work group accessing the same centroid point at a given step of the computation, resulting in a bank-conflict free broadcast from the local memory. All experiments for these results were obtained on a two-dimensional data set with 300 centroids.

Next, we characterized our most optimized K-Means algorithm by varying the different algorithmic parameters. Figure 5(a) presents the performance variation with different number of centroids, as the number of data points increases. Figure 5(b) shows the performance variation with 2D and 4D data sets, each plotted for 100 and 300 centroids. The measurements indicate that K-Means is able to achieve higher performance with higher dimensional data. Finally, Figures 5(c) and 5(d) show that there is no measurable change in performance with the number of iterations.

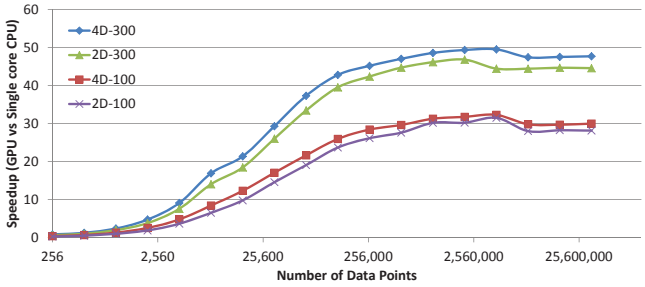
## 4.3 Sharing Work between CPU and GPU

In the OpenCL K-Means implementation, we follow a hybrid approach where cluster assignment step is performed in the GPU and the centroid update step is performed in the CPU. A single kernel thread calculates the centroid assign-

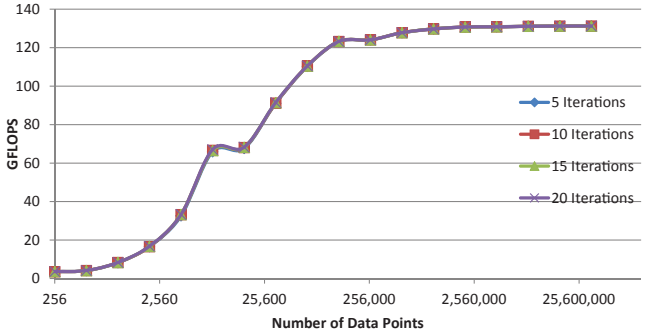
[1]We use a 3 GHz Intel Core 2 Duo Xeon processor, with 4 MB L2 cache and 8 GB RAM, in all our experiments.



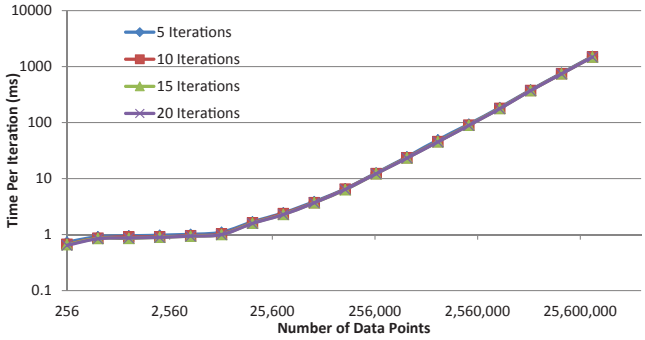

(a) K-Means: varying number of centers.




(b) K-Means: varying number of dimensions.




(c) K-Means: varying number of iterations.




(d) K-Means (per iteration): varying number of iterations.

**Figure 5: K-Means with varying algorithmic parameters.**

ment for one data point. These assignments are then transfered back to the CPU to calculate the new centroid values. While some recent efforts have found that performing all the computation on the GPU can be beneficial, especially, when data sets are large [8], that approach forgoes the opportunity to make use of the powerful CPU cores that might also be available in a distributed environment. Performing partial computation on the CPU allows our approach to implement software pipelining within iteration by interleaving the work partitions and across several iterations through speculation.

## 4.4 Overhead Estimation

In order to isolate the data communication and kernel scheduling overheads we used a simple performance model. Suppose that $c_s$ is the time to perform K-Means computation and $o_s$ is the total overheads, for $s$ data points. Then, the total running time of the algorithm, $T_s$ is given by:

$$T_s = c_s + o_s \quad (1)$$

Suppose that we double the computation that each kernel thread performs. Since the overheads remain more or less unchanged, the total running time, $T'_s$, with double the computation is given by:

$$T'_s = 2{\cdot}c_s + o_s \quad (2)$$

By empirically measuring $T_s$ and $T'_s$ and using Equations 1 and 2, we can estimate the overheads. Figure 6 shows $T'_s$ ("double compute"), $T_s$ ("regular"), $c$ ("compute only") and $o$ ("overhead"). The running times are in seconds (left vertical axis) and overhead is plotted as a percentage of the compute time, $c$ (right vertical axis). Clearly, for small data sets the overheads are prohibitively high. This indicates that, in general, a viable strategy to get the best performance would be to offload the computation on the GPU only when data sets are sufficiently large. Empirically measured parameters can guide the decision process at run time.

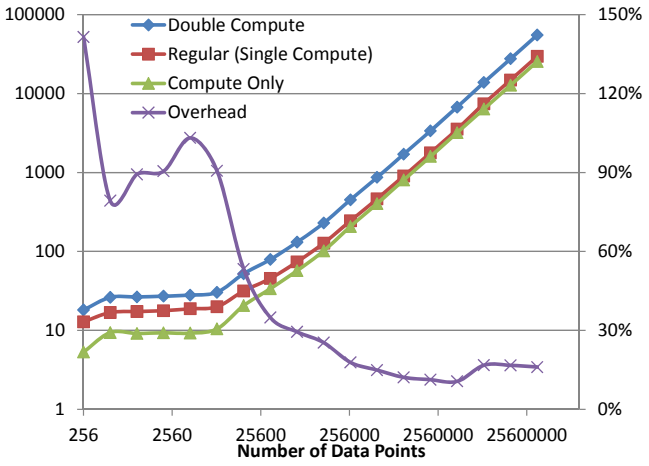



**Figure 6: Overheads in OpenCL KMeans.**

# 5. MDS

The objective of multi-dimensional scaling (MDS) is to map a data set in high-dimensional space to a user-defined lower dimensional space with respect to pairwise proximity of the data points [16, 5]. Dimensional scaling is used mainly in visualization of high-dimensional data by mapping them to two or three dimensional space. MDS has been used to visualize data in diverse domains, including, but not limited to, bio-informatics, geology, information sciences, and marketing.

One of the popular algorithms to perform MDS is Scaling by MAjorizing a COmplicated Function (SMACOF) [7]. SMACOF is an iterative majorization algorithm to solve MDS problem with STRESS criterion, which is similar to expectation-maximization. In this paper, we implement the parallel SMACOF algorithm described by Bae et al [1].

The input for MDS is an $N{\times}N$ matrix of pairwise proximity values, where $N$ is the number of data points in the high-dimensional space. The resultant lower dimensional mapping in $D$ dimensions, called the `X` values, is an $N{\times}D$ matrix. For the purposes of this paper, we performed an unweighted mapping resulting in two main steps in the algorithm: (a) calculating new `X` values, and (b) calculating the stress of the new `X` values. There needs to be a global barrier between the two steps as stress value calculation requires all of the new `X` values. However the reduction step for `X` values in MDS is much simpler than in K-Means. Since each data point, $k$, independently produces the value `X[k]`, the reduction step reduces to simple aggregation in memory. Figure 7 outlines our OpenCL implementation of MDS.

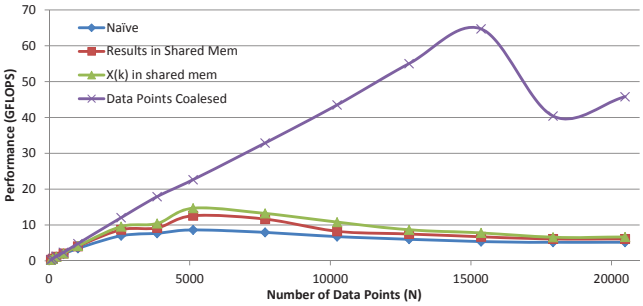
The number of floating pointer operations, $F$, per iteration per thread is given by $F = (8DN + 7N + 3D + 1)$, resulting in a total of $F{\times}N{\times}I$ floating point operations per calculation, where $I$ is the number of iterations, $N$ is the number of data points, and $D$ is the dimensionality of the lower dimensional space.

```
__kernel MDS(__global float* data,
  __global float* x,__global float* newX){

  gid = get_global_id(0);

  for (int j = 0; j < WIDTH ; j++)
  {
    distance  = dist(x[gid][], x[j][]);
    bofZ = k * (data[gid][j] /distance);
    rowSum += bofZ;
    newX[gid][] += bofz * x[j][];
  }

  newX[gid][] += k * rowSum * x[gid][];
  newX[gid][] = newX[gid][]/WIDTH;

  barrier(CLK_GLOBAL_MEM_FENCE);

  __local float sigma[];
  for (int j = 0; j < WIDTH; j++)
  {
    distance = dist(newX[gid][], newX[j][])
    sigma[lid] += (data[gid][j]-distance)^2;
  }

  stress = hierachicalReduction(sigma[]);
}
```

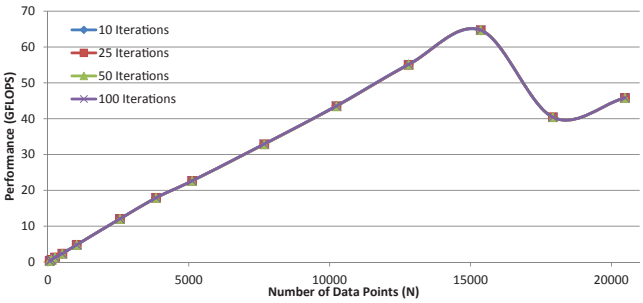
**Figure 7: Outline of MDS in OpenCL.**

## 5.1 Leveraging Local Memory

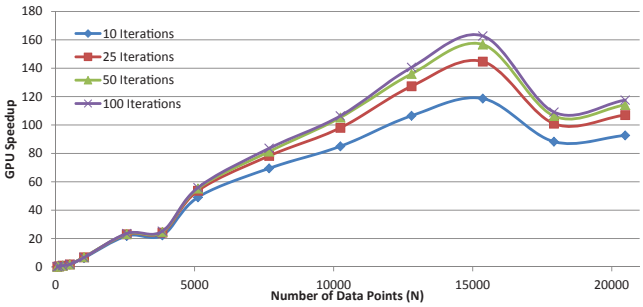
In a naïve implementation all the data points, `X` values, and result (new `X` values) are stored in global memory. SMA-




(a) MDS performance with the different optimizations steps.




(b) MDS: varying number of iterations.




(c) MDS per iteration: varying number of iterations.

**Figure 8: MDS with varying algorithmic parameters.**

COF MDS algorithm uses a significant number of temporary runtime matrices for intermediate data storage. We restructured the algorithm to eliminate the larger temporary run time matrices, as they proved to be very costly in terms of space as well as performance. The kernel was redesigned to process a single row at a time.

After eliminating the run time data structures, `X'[k][]` matrix points were used in several locations of the algorithm to store intermediate results, which were stored in local memory and copied to global memory only at barrier synchronization. As an added advantage, we were able to reuse the intermediate values in local memory when calculating the stress values. This resulted in a large performance improvement (up to about 45%) for intermediate size inputs.

`X[k]` values for each thread `k` were copied to local memory before the computation. `X` values belonging to the row that is being processed by the thread gets accessed many more times compared to the other `X` values. Hence, copying these `X` values to local memory turns out to be worthwhile.

## 5.2 Optimizing Memory Access

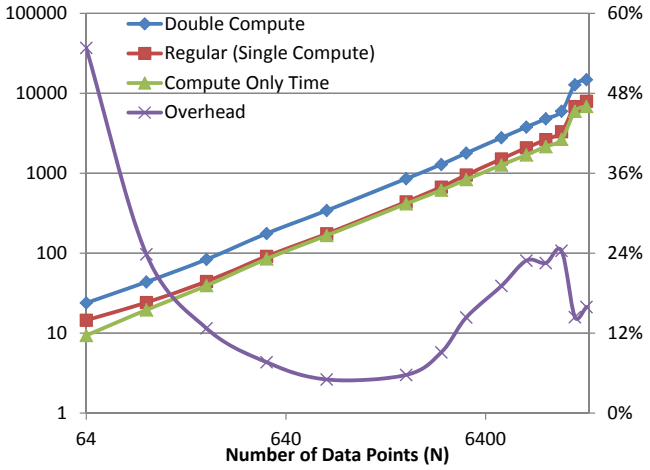



**Figure 9: Overheads in OpenCL MDS.**

All data points belonging to the data row that a thread is processing are iterated through twice inside the kernel. We encourage hardware coalescing of these accesses by storing the data in global memory in column-major format, which causes contiguous memory access from threads inside a local work group. Figure 8(a) shows that data placement to encourage hardware coalescing results in a significant performance improvement.

Similarly we experimented with storing the `X` values in column-major format, but it resulted in a slight performance degradation. The access pattern for the `X` values is different from that for the data points. All the threads in a local work group access the same `X` value at a given step. As we noted in Section 4.2, we observe a similar behavior with the K-Means clustering algorithm.

Performance improvements resulting from each of the above optimizations are summarized in Figure 8(a). Unfortunately, we do not yet understand why the performance drops suddenly after a certain large number of data points and then begins to improve again. Possible explanations could include increased data bus contention, or memory bank conflicts. However, we would need more investigation to determine the exact cause. Figures 8(b) and 8(c) show performance numbers with varying number of iterations, which show similar trends.

## 5.3 Sharing Work between CPU and GPU

In the case of MDS, there is not a good case for dividing the work between CPU and GPU. In our experiments, the entire computation was done on the GPU. On the other hand, as the measured overheads show below, certain problem sizes might be better done on the CPU.

## 5.4 Overhead Estimation

Following the model that was used for K-Means in Section 4.4, we performed similar experiments for estimating kernel scheduling and data transfer overheads in MDS. Figure 9 shows the results. As in K-Means, we note that the overheads change with the input data size. In the case of MDS, however, there are two useful cutoffs, one for small data sizes and another for large data sizes—on either ends overheads become high and the computation might achieve

higher performance on the CPU if the data have to be transferred from the CPU memory, which is what we have assumed in the overhead computations.

# 6. PAGERANK

PageRank algorithm, developed by Page and Brin [6], analyzes linkage information of a set of linked documents to measure the relative importance of each document whithin the set. PageRank of a certain document depends on the number and the PageRank of other documents linked to it.

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} \quad (3)$$

Equation 3 defines PageRank, where $\{p_1, \ldots, p_N\}$ is the set of documents, $M(p_i)$ is the set of documents that link to $p_i$, $L(p_j)$ is the number of outbound links on $p_j$, and $N$ is the total number of pages. PageRank calculation can be performed using an iterative power method, resulting in the multiplication of a sparse matrix and a vector. The linkage graph for the web is very sparse and follows a power law distribution [2], presenting unique implementation challenges for PageRank.

For our OpenCL PageRank implementation we used a modified compressed sparse row (CSR) format and modified ELLPACK format [4] to store the matrix representing the link graph. Typically the sparse matrix used for PageRank stores $1/L(p_j)$ in an additional *data* array. We eliminated the data array by storing the intermediate page rank values as $PR(p_j)/L(p_j)$, significantly reducing memory usage and accesses. We made a similar modification to ELLPACK format. We preprocessed and used the Stanford web data set from the Stanford Large Network Dataset [25] for our experiments.

```
__kernel PageRankCSR(__global float* pointers,
  __global float* indices,__global float* x,
  __global float* newX){

  gid = get_global_id(0);

  start = pointers[(gid)];
  end = pointers[(gid)+1];

  for (int i=start; i < end ; i++)
  {
    newRank += ranks[indices[i]];
  }

  newRank = ((1-d)/numPages) + (d * newRank);
  // To avoid storing 1/L(pj) in the matrix.
  newRanks[gid] = newRank/numPages;
}
```

**Figure 10: Outline of PageRank (CSR) in OpenCL.**

## 6.1 Leveraging Local Memory

We were not able to utilize local memory to store all the data in the GPU kernel due to the variable sizes of matrix rows and the large size of the PageRank vector. However, we used local memory for data points in the ELLPACK kernel.

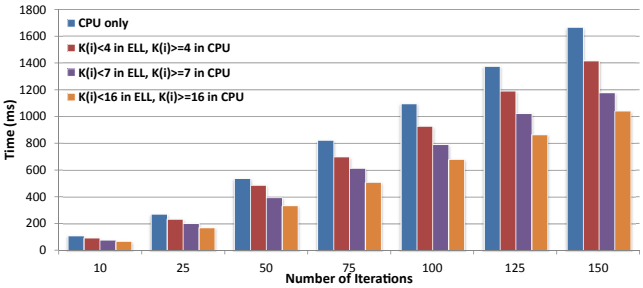



**Figure 11: Potential implementations of PageRank.**

## 6.2 Optimizing Memory Access

Due to the irregular memory access pattern arising out of indirect array accesses, sparse matrix vector computation is not amenable to memory access optimizations. However, the index array, especially in the ELLPACK format, is stored in appropriate order to enable contiguous memory accesses.

## 6.3 Sharing Work between CPU and GPU

Due to the power law distribution of non-zero elements, a small number of rows contains a large number of elements, but a large number of rows are very sparse. In a preprocessing step, the rows are partitioned into two or more sets of those containing a small number of elements and the remainder containing higher number of elements. The more dense rows could be computed either on the CPU or the GPU using the CSR format directly. The rows with smaller number of non-zero elements are reformatted into the ELLPACK format and computed on the GPU. We evaluated several partitioning alternatives, shown in Figure 11.

The leftmost bars represent the running times on CPU. The next three bars represents computing all rows with greater than or equal to $k$ elements on the CPU, where $k$ is 4, 7, and 16, respectively. The rows with fewer than $k$ elements are transformed into ELLPACK format and computed on the GPU. Moreover, when $k = 7$, two distinct GPU kernels are used, one for computing rows with up to 3 elements and another for computing rows with 4 to 7 elements. Similarly, for $k = 16$, an additional third kernel is used to process rows with 8 to 15 elements. Splitting the kernels not only improves the GPU occupancy, but also allows those kernels to be executed concurrently.

In Figure 11 we do not include the overheads of the linear time preprocessing step and of host-device data transfers, both of which are relatively easy to estimate. However, we also do not assume any parallelism between the multiple kernels processing the rows in ELLPACK format. Our main observation from these experiments is that sharing work between CPU and GPU for sparse matrix-vector multiplication is a fruitful strategy. Moreover, unlike previous attempts recommending hybrid matrix representation that used a single kernel for the part of the matrix in ELLPACK format [4], our experiments indicate that it is beneficial to use multiple kernels to handle rows with different numbers of non-zero elements. The problem of deciding the exact partitioning and the exact number of kernels is outside the scope of this paper and we leave that as part of future work.

Instead of computing the matrix partition with denser rows on the CPU, it could also be computed on the GPU. We

also implemented a sparse matrix-vector product algorithm using CSR representation on the GPU (not shown in the figure). Our experiments indicate that GPU can take an order of magnitude more time for that computation than CPU, underlining the role of CPU for certain algorithm classes.

# 7. LESSONS

In this study we set out to determine if we could characterize some core data processing statistical kernels for commonly used optimization techniques on GPUs. We focused on three widely used kernels and three important optimizations. We chose to use OpenCL, since there are fewer experimental studies on OpenCL, compared to CUDA, and the multi-platform availability of OpenCL would allow us to extend our research to other diverse hardware platforms.

### *Leveraging Local Memory.*

It is not surprising that making use of faster local memory turns out to be one of the most important optimizations within OpenCL kernels. In many cases, decision about which data to keep in local memory is straightforward based on reuse pattern and data size. For example, in K-Means and MDS it is not possible to keep the entire data set in local memory, since it is too big. However, the centroids in K-Means and intermediate values in MDS can be fruitfully stored there. Unfortunately, in some cases, such as portions of MDS, leveraging local memory requires making algorithmic changes in the code, which could be a challenge for automatic translators.

### *Optimizing Memory Access.*

Laying out data in memory is a known useful technique on CPUs. On GPUs, we observed mixed results. While data layout in local memory turned out to be useful for K-Means and not for MDS, layout in global memory had significant impact on MDS and no observable impact on K-Means. This behavior is likely a result of different memory access patterns. In general, contiguous global memory accesses encourage hardware coalescing, whereas on local memory bank conflicts play a more critical role. Thus, the two levels of memories require different layout management strategies. However, as long as the memory access patterns are known the benefits are predictable, thus making this optimization amenable to automatic translation.

### *Sharing work between CPU and GPU.*

One major issue in sharing work between CPU and GPU is the host-device data transfers. Clearly, this has to be balanced against the improved parallelism across GPUs and multi-core CPUs. Moreover, within the context of our study, there is also the issue of how data across nodes get transferred. If the data must move through CPU memory then in certain cases it might be beneficial to perform the computation on the CPU. Through our simple performance model and the overhead graphs the trade-offs are apparent. These graphs could also help in determining the cutoffs where offloading computation on the GPU is worthwhile. Finally, in iterative algorithms, where kernels are invoked repeatedly, offloading part of the computation on the GPUs can also enable software pipelining between CPU and GPU interleaving different work partitions.

Another factor in determining the division of work is the complexity of control flow. For instance, a reduction operation in K-Means, or a sparse matrix-vector multiply with relatively high density of non-zero values that might involve a reduction operation, may be better suited for computing on the CPU. This would be especially attractive if there is sufficient other work to overlap with GPU computations.

Finally, the differences in precision between CPU and GPU can sometimes cause an iterative algorithm to require different number of iterations on the two. A decision strategy for scheduling an iterative algorithm between CPU and GPU may also need to account for these differences.

Unlike the previous two optimizations, the value of this one is determined largely by the nature of input data. As a result, a dynamic mechanism to schedule computation just-in-time based on the category of input could be a more useful strategy than a static one.

### *Caching of loop-invariant data.*

Transferring of input data from CPU memory to GPU memory is a major cost of performing data intensive statistical computations on GPU. The speedup from GPU over CPU should be large enough to mitigate this initial data transfer cost. However, statistical iterative algorithms have the ability to reuse the loop-invariant input data in GPU memory across iterations avoiding significant data transfer costs. In order to harness this benefit, the loop-invariant data should fit in to the GPU global memory and should be retained throughout the computation. When the size of loop-invariant data is larger than the GPU global memory, it is advantageous to distribute the work across compute nodes rather than swapping the data in and out of the GPU memory.

### *OpenCL experience.*

OpenCL provides a flexible programing environment and supports simple synchronization primitives, which helps in writing substantial kernels. However, details such as the absence of debugging support and lack of dynamic memory allocation still make it a challenge writing code in OpenCL. One possible way to make OpenCL-based GPU computing accessible to more users is to develop compilers for higher level languages that target OpenCL. Insights gained through targeted application studies, such as this, could be a useful input to such compiler developers.

# 8. RELATED WORK

Emergence of accessible programming interfaces and industry standard languages has resulted in tremendously increased interest in using GPUs for general purpose computing. CUDA, by NVIDIA, has been the most popular framework for this purpose [21]. In addition to directly studying application implementations in CUDA [11, 23], there have been recent research projects exploring CUDA in hybrid CUDA/MPI environment [22], and using CUDA as a target in automatic translation [18, 17, 3].

There have been several past attempts at implementing the K-Means clustering algorithm on GPUs, mostly using OpenGL or CUDA [24, 12, 27, 19, 15]. Recently, the K-Means algorithm has also been implemented using OpenCL by Dhanasekaran et al [8]. In contrast to the approach of Dhanasekaran et al., who implemented the reduction step on GPUs in order to handle very large data sets, we chose to

mirror the earlier efforts with CUDA and perform the reduction step on the CPU. Even though that involves transferring the reduction data to CPU, we found that the amount of data that needed to be transferred was relatively small. In optimizing K-Means, we used the device shared memory to store the map data. As a result, when dealing with very large data sets, which motivated Dhanasekaran et al.'s research, our optimized kernel would run out of shared memory before the reduction data becomes too large to become a bottleneck. Further research is needed to determine the trade-offs of giving up the optimization of device shared-memory and performing the reduction on the GPU.

We implemented the MDS kernel based on an interpolation algorithm by Bae et al. [1]. Glimmer is another multilevel MDS implementation [13]. While Glimmer implements multilevel MDS using OpenGL Shading Language (GLSL) for large data sets, we used an interpolated approach to limit the data size, which has been found to be useful in certain contexts. This allowed us to experiment with optimizing the algorithm for realistic contexts, without worrying about dealing with data sets that do not fit in memory.

The computationally intensive part of PageRank is sparse matrix-vector multiplication. We followed the guidelines from an NVIDIA study for implementing the sparse matrix-vector multiplication [4]. The sparse matrix in PageRank algorithm usually results from graphs following power law. Recent efforts to optimize PageRank include using a low-level API to optimize sparse matrix-vector product by using the power law characteristics of the sparse matrix [28]. More recently, Yang et al. leveraged this property to auto-tune sparse matrix-vector multiplication on GPUs [29]. They built an analytical model of CUDA kernels and estimated parameters, such as tile size, for optimal execution.

# 9. CONCLUSION AND FUTURE WORK

We have presented an experimental evaluation of three important kernels used in iterative statistical applications for large scale data processing, using OpenCL. We evaluated three optimization techniques for each, based on leveraging fast local memory, laying out data for faster memory access, and dividing the work between CPU and GPU. We conclude that leveraging local memory is critical to performance in almost all the cases. Data layout is important in certain cases, but when it is, it has significant impact. In contrast to other optimizations, sharing work between CPU and GPU may be input data dependent, as in the case of K-Means, which points to the importance of dynamic just-in-time scheduling decisions.

Our planned future work includes extending the kernels to a distributed environment, which is the context that has motivated our study. Other possible directions include comparing the OpenCL performance with CUDA, studying more kernels from, possibly, other domains, and exploring more aggressive CPU/GPU sharing on more recent hardware that has improved memory bandwidth.

# 10. REFERENCES


[1] S.-H. Bae, J. Y. Choi, J. Qiu, and G. C. Fox. Dimension reduction and visualization of large high-dimensional data via interpolation. In *Proceedings of the 19th ACM International Symposium on HIgh Performance Distributed Computing (HPDC)*, pages 203–214, 2010. DOI: 10.1145/1851476.1851501.

[2] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, Oct. 1999. DOI: 10.1126/science.286.5439.509.

[3] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *In Proceedings of the 19th International Conference on Compiler Construction (CC)*, pages 244–263, 2010. DOI: 10.1007/978-3-642-11970-5.

[4] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[5] I. Borg and P. J. F. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Statistics. Springer, second edition, 2005.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the Seventh International World Wide Web Conference*, volume 30 of *Computer Networks and ISDN Systems*, pages 107–117, Apr. 1998. DOI: 10.1016/S0169-7552(98)00110-X.

[7] J. de Leeuw. Convergence of the majorization method for multidimensional scaling. *Journal of Classification*, 5(2):163–180, 1988. DOI: 10.1007/BF01897162.

[8] B. Dhanasekaran and N. Rubin. A new method for GPU based irregular reductions and its application to k-means clustering. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4)*, 2011. DOI: 10.1145/1964179.1964182.

[9] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance and Distributed Computing (HPDC)*, pages 810–818, 2010. DOI: 10.1145/1851476.1851593.

[10] K. Group. OpenCL: The open standard for parallel programming of heterogeneous systems. On the web. http://www.khronos.org/opencl/.

[11] T. R. Halfhill. Parallel processing with CUDA. *Microprocessor Report*, Jan. 2008.

[12] B. Hong-tao, H. Li-li, O. Dan-tong, L. Zhan-shan, and L. He. K-Means on commodity GPUs with CUDA. In *Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering*, 2009. DOI: 10.1109/CSIE.2009.491.

[13] S. Ingram, T. Munzner, and M. Olano. Glimmer: Multilevel MDS on GPU. *IEEE Transactions on Visualization and Computer Graphics*, 15(2):249–261, 2009. DOI: 10.1109/TVCG.2008.85.

[14] M. Johansson and O. Winter. General purpose computing on graphics processing units using OpenCL. Masters thesis, Chalmers University of Technology, Department of Computer Science and Engineering Göteborg, Sweden, June 2010.

[15] K. J. Kohlhoff, M. H. Sosnick, W. T. Hsu, V. S. Pande, and R. B. Alteman. CAMPAIGN: An open-source library of GPU-accelerated data clustering algorithms. *Bioinformatics*, 2011. DOI: 10.1093/bioinformatics/btr386.

[16] J. B. Kruskal and M. Wish. *Multidimensional Scaling*. Sage Publications Inc., 1978.

[17] S. Lee and R. Eigenmann. OpenMPC: Extended

OpenMP programming and tuning for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2010)*, 2010. DOI: 10.1109/SC.2010.36.

[18] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009. DOI: 10.1145/1504176.1504194.

[19] Y. Li, K. Zhao, X. Chu, and J. Liu. Speeding up K-Means algorithm by GPUs. In *Proceedings of the 2010 IEEE 10th International Conference on Computer and Information Technology (CIT)*, pages 115–122, 2010. DOI: 10.1109/CIT.2010.60.

[20] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, Mar. 1982. DOI: 10.1109/TIT.1982.1056489.

[21] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide*, version 1.1 edition, Nov. 2007.

[22] S. J. Pennycook, S. D. Hammond, S. A. Jarvis, and G. Mudalige. Performance analysis of a hybrid MPI/CUDA implementation of the NAS-LU benchmark. In *Proceedings of the First International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS)*, 2010.

[23] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 73–82, 2008. DOI: 10.1145/1345206.1345220.

[24] S. A. A. Shalom, M. Dash, and M. Tue. Efficient K-Means clustering using accelerated graphics processors. In *Proceedings of the 10th International Conference on Data Warehousing and Knowledge Discovery (DaWak)*, volume 5182 of *Lecture Notes in Computer Science*, pages 166–175, 2008. DOI: 10.1007/978-3-540-85836-2_16.

[25] SNAP. Stanford network analysis project. On the web. http://snap.stanford.edu/.

[26] L. G. Valiant. Bulk-synchronous parallel computers. In M. Reeve, editor, *Parallel Processing and Artificial Intelligence*, pages 15–22. John Wiley & Sons, 1989.

[27] R. Wu, B. Zhang, and M. Hsu. GPU-accelerated large scale analytics. Technical Report HPL-2009-38, Hewlett Packard Lts, 2009.

[28] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu. Efficient PageRank and SpMV computation on AMD GPUs. In *Proceedings of the 39th International Conference on Parallel Processing (ICPP)*, pages 81–89, 2010. DOI: 10.1109/ICPP.2010.17.

[29] X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining. *Proceedings of the VLDB Endowment*, 4(4):231–242, 2011.