Iterative Statistical Kernels on Contemporary GPUs

Thilina Gunarathne

School of Informatics and Computing
Indiana University, Bloomington, IN 47405, USA
Email: tgunarat@cs.indiana.edu

Bimalee Salpitikorala

School of Informatics and Computing
Indiana University, Bloomington, IN 47405, USA
Email: ssalpiti@cs.indiana.edu

Arun Chauhan

School of Informatics and Computing
Indiana University, Bloomington, IN 47405, USA
Email: achauhan@cs.indiana.edu

Geoffrey Fox

School of Informatics and Computing
Indiana University, Bloomington, IN 47405, USA
Email: gcf@cs.indiana.edu

Abstract:

We present a study of three important kernels that occur frequently in iterative statistical applications: Multi-Dimensional Scaling (MDS), PageRank, and K-Means. We implemented each kernel using OpenCL and evaluated their performance on NVIDIA Tesla and NVIDIA Fermi GPGPU cards using dedicated hardware, and in the case of Fermi, also on the Amazon EC2 cloud-computing environment. By examining the underlying algorithms and empirically measuring the performance of various components of the kernels we explored the optimization of these kernels by four main techniques: (1) caching invariant data in GPU memory across iterations, (2) selectively placing data in different memory levels, (3) rearranging data in memory, and (4) dividing the work between the GPU and the CPU. We also implemented a novel algorithm for MDS and a novel data layout scheme for PageRank. Our optimizations resulted in performance improvements of up to 5X to 6X, compared to naïve OpenCL implementations and up to 100X improvement over single-core CPU. 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.

Keywords: GPUs; OpenCL; MDS; multi-dimensional-scaling; PageRank; kmeans clustering; iterative statistical applications; cloud GPUs; sparse matrix-vector multiplication.

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 [27]. 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 important algorithms, which we refer to as *kernels*, that find use in such iterative statistical applications. (a) Multi-Dimensional Scaling (MDS), (b) PageRank and (c) K-Means Clustering. MDS is a statistical technique used for visualizing and exploring large data sets in high-dimensional spaces by mapping them in to lower dimensional spaces. In this paper, we implement the Scaling by MAjorizing a COmplicated Function (SMACOF) [7] MDS algorithm using OpenCL by utilizing the parallelization methods described by Bae et al [1]. Our OpenCL SMACOF MDS implementation was able to perform up to 180 times faster than a sequential implementation on an Intel Core i7 (3 Ghz) CPU.

PageRank [6] is an iterative link analysis algorithm that analyzes linkage information of a set of linked documents, such as web pages, to measure the relative importance of each document within the set. Core computation of PageRank relies on a sparse matrix-vector multiplication, where the sparse-matrix can often be very unstructured with the number of non-zero elements in rows observing distributions such as Power Law, which makes these computations very challenging to perform in GPUs. In this paper we explore several traditional mechanisms and then introduce a novel data storage format optimized for processing power-law distributed sparse matrix-vector multiplications. Our new format requires only minimal pre-processing to generate and the associated algorithm performed several times faster than the traditional methods and the sequential versions. K-Means Clustering is a clustering algorithm used in many machine learning applications. K-Means Clustering is one of the most popular and well-known kernels in the iterative scientific applications category, which we use as an important representative to guide GPGPU optimizations for this class of applications.

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. Caching invariant data in GPU memory to be used across kernel invocations (i.e., algorithm iterations);
2. Utilizing OpenCL *local memory*, by software-controlled caching of selected data;
3. Reorganizing data in memory, to encourage hardware-driven memory access coalescing or to avoid bank conflicts; and
4. Dividing the computation between CPUs and GPUs, to establish a software pipeline across iterations.

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. We compare the performance of these applications on two generations of NVIDIA GPGPU processors and on Amazon EC2 GPU compute instances for both single precision as well as double precision computations. We show that cloud GPU instances provide performance comparable to non-cloud GPU nodes for almost all of our kernels—except for single-precision K-Means Clustering, which showed a 20% performance degradation in the cloud environment—making GPUs in the cloud a viable choice for these kernels. The speedup of GPU computations over the corresponding sequential CPU computations is usually much better in cloud.

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.

The contributions of the paper include: (1) Extensive experimental evaluation of three popular statistical computing kernels on two generations of NVIDIA GPUs in a dedicated as well as cloud-computing environment; (2) First GPU implementation of the relatively more accurate SMACOF algorithm for multi-dimensional scaling; (3) A novel data layout scheme to improve the performance of matrix-vector product on GPUs, which does not rely on inverse power-law distribution of matrix row lengths (hence, is more broadly applicable); and (4) A systemic study of the three kernels for memory hierarchy optimizations on the two GPU cards.

2 Background

Boosted by the growing demand for gaming power, the traditional fixed function graphics pipeline of GPUs have evolved into a full-fledged programmable hardware chain [15].

2.1 OpenCL

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.

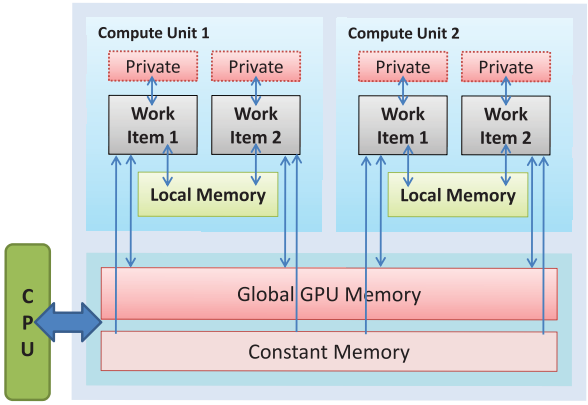


Figure 1 OpenCL memory hierarchy. In the current NVIDIA OpenCL implementation, private memory is physically located in global memory.

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.

2.2 NVIDIA Tesla GPGPU's

In this paper we use NVIDIA Tesla C1060, Tesla "Fermi" M2050 and Tesla "Fermi" C2070 GPGPUs for our experiments. Tesla C1060 consists of 240 processor cores and 4GB GPU global memory with 102 GB/sec peak memory bandwidth. It has a theoretical peak performance of 622 GFLOPS (933 GFLOPS for special operations) for single precision computations and 78 GFLOPS for double precision computations.

NVIDIA M2050 and C2070 are based on the "Fermi" architecture with better support for double precision computations than the Tesla C1060. These GPGPUs also feature L1 and L2 caches and ECC memory error protection. Tesla M2050 and C2070 contain 448 processor cores resulting in a theoretical peak performance of 1030 GFLOPS (1288 GFLOPS for special operations) for single precision computations and 515 GFLOPS for double precision computations. Tesla M2050 and C2070 have 3 GB and 6 GB GPU global memory, respectively, with 148.4 GB/sec and 144 GB/sec peak memory bandwidths.

2.3 Amazon EC2 Cluster GPU instances

Amazon Web Services pioneered the commercial cloud service offerings and currently provide a rich set of on demand compute, storage and data communication services including but not limited to Amazon Elastic Compute Cloud (EC2), Simple Storage Service (S3) and Elastic Map Reduce (EMR). Amazon EC2 provides the users with the ability to dynamically provision hourly built virtual instances with a variety of configurations and environments, giving the users the ability to obtain a dynamically re-sizable virtual cluster in a matter of minutes. EC2 instance types of interest to high performance computing include the Cluster Compute and Cluster GPU instances, which provide high CPU and network performance required for the HPC applications.

The single EC2 Cluster GPU instance contains two NVIDIA Tesla "Fermi" M2050 GPGPUs with 3 GB GPU memory in each GPGPU, two Intel Xeon X5570 quad-core "Nehalem" processors and 22 GB of RAM. Two GPGPUs in the Cluster GPU instance offer a combined theoretical maximum double precision performance of 1030 GFLOPS and more than 2060 GFLOPS theoretical single precision performance. The current hourly price of an on-demand cluster GPU instance is $2.10. However, Amazon EC2 also provides spot instances, which allow the users to bid on unused Amazon EC2 capacity and run those instances for as long as the bid exceeds the current spot price. Throughout our testing for this paper we used EC2 cluster GPU spot instances at the rate of $0.65 per hour, which converts to $0.65 per hour of one TFLOPS of double precision performance (or ≈$0.33 per hour of one TFLOPS of single precision performance) from the GPGPUs.

2.4 Performance Testing

In this paper we analyzed and compared the performance of OpenCL application implementations in the environments given in Table 1. We used OpenCL 1.1 and the NVIDIA driver version 290.10 in all the environments. We used the OpenCL event API to obtain the kernel execution time for applications and used it for the calculation of GFLOPS. Total run time, including the time for initial data copy from CPU memory to GPU memory and the time to copy results back, was used for all the other graphs.

3 Iterative Statistical Applications

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

Table 1 GPGPU environments used in this paper

Name	*GPU*	*GPU Memory*	*Peak Performance (GFLOPS)*	*GPU-Mem Bandwidth*	*CPU*	*RAM*
Mac Pro	NVIDIA Tesla C1060	4GB	77 (DP) 622(SP)	102 GB/sec	Intel Xeon X5365 (3.00GHz)	8GB
Fermi	NVIDIA Tesla C2070	6GB	515 (DP) 1030 (SP)	144 GB/sec	Intel Core i7-950 (3.07GHz)	8GB
EC2 Fermi (VM)	NVIDIA Tesla M2050	3GB	515 (DP) 1030 (SP)	148 GB/sec	Intel Xeon X5570 (2.93GHz)	22GB

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] and Twister4Azure [11], 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 we use global memory for this broadcast. Even though constant memory could potentially result in better performance, it is often too small to hold the loop-variant delta for the MDS and PageRank kernels we studied.

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. 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.

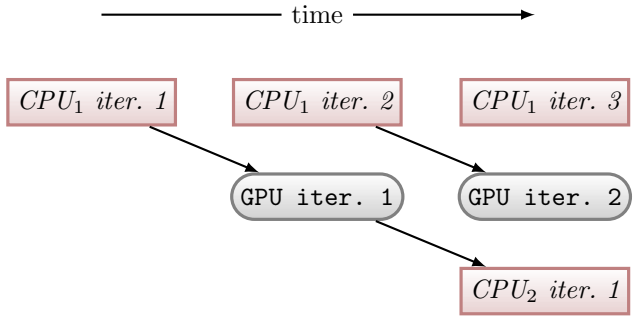


Figure 2 Software pipelining to leverage GPUs for loop-level parallelism.

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. It should be noted that even though software pipelining served as a motivating factor in designing our algorithms, we did not use software pipelining for the kernels used in this study.

4 MDS

Multi-dimensional scaling (MDS) is a widely used statistical mechanism used 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 [17, 5]. Multi-dimensional scaling is used mainly to explore large data sets in high-dimensional spaces by visualizing them 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]. To the best of our knowledge, this is the first GPU implementation of the parallel SMACOF algorithm. The pairwise proximity data input for MDS is an $N \times N$ matrix of pairwise dissimilarity (or similarity) values, where N is the number of data points in the high-dimensional space. The resultant lower dimensional mapping in the target (D) dimension, called the `X` values, is an $N \times$D matrix, in which each row represent the data points in the lower dimensional space. The core of the SMACOF MDS algorithm consists of iteratively calculating new `X` values and the performing the stress value calculation.

For the purposes of this paper, we performed an unweighted (weight = 1) mapping resulting in two main steps in the algorithm: (a) calculating new `X` values (`BCCalc`), and (b) calculating the stress of the new `X` values. A global barrier exists between the above two steps as stress value calculation requires all of the new `X` values. From inside a kernel, OpenCL APIs only supports synchronization within a work-group and requires the computations to be broken into separate kernels to achieve global synchronization. Hence, we implemented OpenCL MDS implementation using separate kernels for `BCCalc` and `StressCalc` as shown in pseudo-code in Figure 3. These two kernels are iteratively scheduled one after another till the STRESS value criterion is reached for the computation.

```
#pragma OPENCL EXTENSION cl_khr_fp64:enable

__kernel MDSBCCalc(double* data,
                   double* x, double* newX) {
  gid = get_global_id(0);
  lid = get_local_id(0);

  // copying some x to shared local mem
  __local lX[lid][] = x[gid][];
  barrier(CLK_LOCAL_MEM_FENCE);

  for (int j = 0; j < WIDTH ; j++)
  {
    distance  = euc_dist(lX[lid][], x[j][]);
    bofZ = k * (data[gid][j] /distance);
    rowSum += bofZ;
    privX[] += bofZ * x[j][];
  }

  privX[gid][] += k * rowSum * lX[lid][];
  newX[gid][] = privX[gid][]/WIDTH;
}

__kernel MDSStressCalc(double* data,
                   double* x, double* newX) {
  for (int j = 0; j < WIDTH; j++)
  {
    distance = dist(newX[gid][], newX[j][])
    sigma += (data[gid][j]-distance)^2;
  }

  stress = hierachicalReduction(sigma);
}
```

Figure 3 Outline of MDS in OpenCL.

The number of floating pointer operations in MDS, 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.

4.1 Optimizations

4.1.1 Caching Invariant Data

MDS has loop-invariant data that can fit in available global memory and that can be reused across iterations. Figure 4 summarizes the benefits of doing that for MDS.

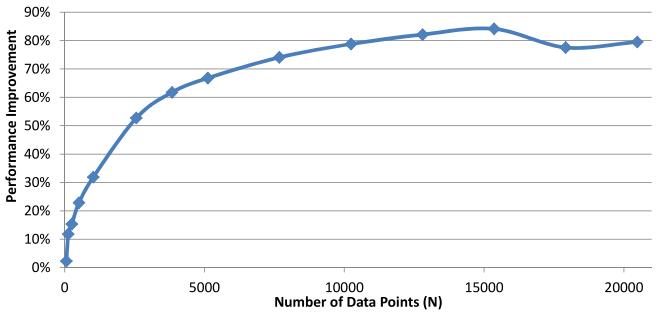


Figure 4 Performance improvement in MDS due to caching of invariant data in GPU memory.

4.1.2 Leveraging Local Memory

In a naïve implementation all the data points (`X` values) and result (new `X` values) are stored in global memory. Parallel SMACOF MDS algorithm by Bae et al. [1] uses a significant number of temporary matrices for intermediate data storage. We restructured the algorithm to eliminate the larger temporary matrices, as they proved to be very costly in terms of space as well as performance. The kernel was redesigned to process a single data row at a time.

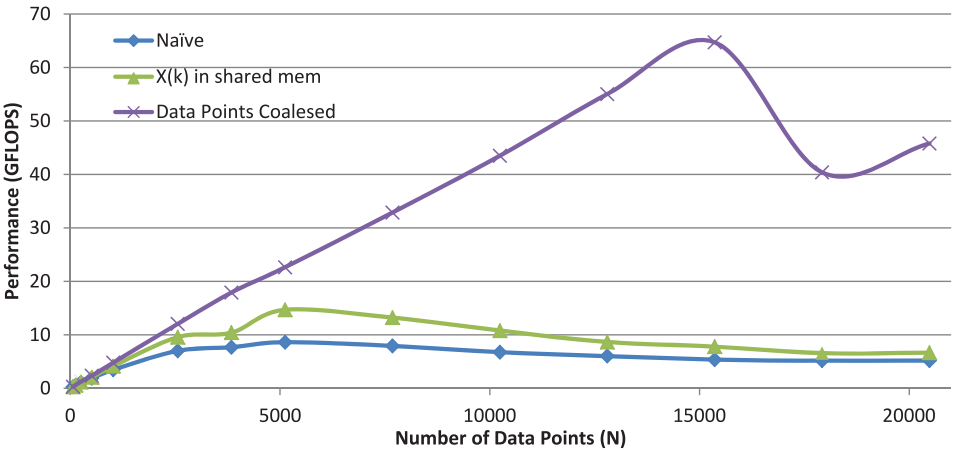
`X[k]` values for each thread data point `k` were copied to local memory before the computation. `X` values belonging to the row that is being processed by the thread get accessed many more times compared to the other `X` values. Hence, copying these `X` values to local memory turns out to be worthwhile. "X(k) in shared mem" curve of Figure 5(a) quantifies the gains.

4.1.3 Optimizing Memory Access

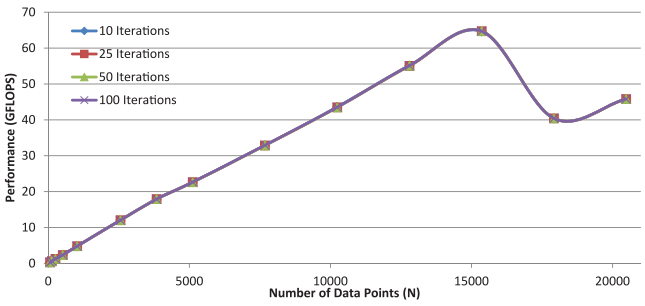
All data points belonging to the data row that a thread is processing are iterated through twice inside the kernel. We encourage hardware coalescing of these accesses by storing the data in global memory in column-major format, which causes contiguous memory access from threads inside a local work group. Figure 5(a) shows that data placement to encourage hardware coalescing results in a significant performance improvement.

We also 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 6.2.2, we observe a similar behavior with the K-Means clustering algorithm.

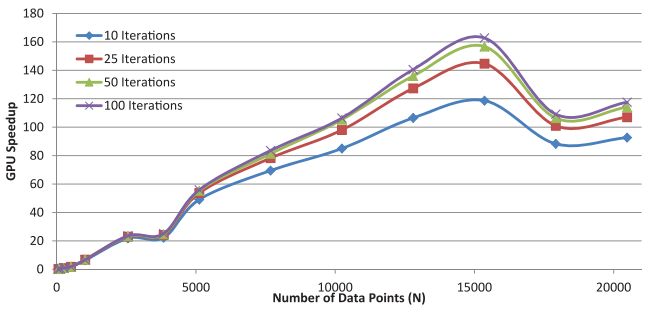
Performance improvements resulting from each of the above optimizations are summarized in Figure 5(a). Unfortunately, we do not yet understand why the performance drops suddenly after a certain large number of data points (peaks at 900 MB data size and drops at 1225 MB data size) and then begins to improve again. Possible explanations could include increased data bus contention, or memory bank conflicts. However, we



(a) MDS performance with the different optimizations steps.



(b) MDS: varying number of iterations.



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

Figure 5 MDS with varying algorithmic parameters.

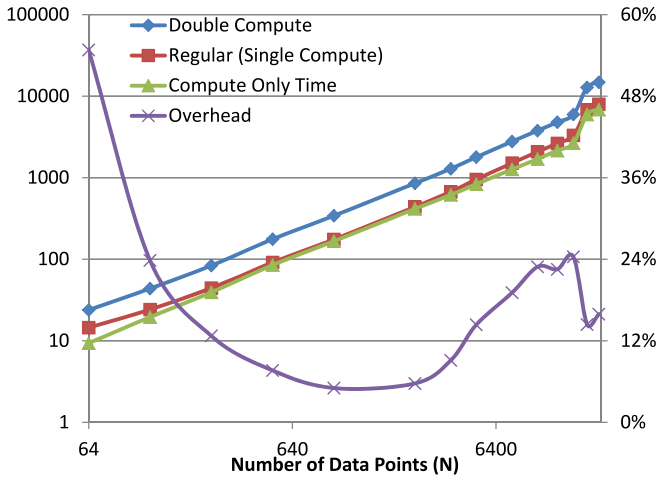


Figure 6 Overheads in OpenCL MDS.

would need more investigation to determine the exact cause. Figures 5(b) and 5(c) show performance numbers with varying number of iterations, which show similar trends.

4.1.4 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.

4.2 Overhead Estimation

We used a simple performance model in order to isolate the overheads caused by data communication and kernel scheduling. Suppose that c_s is the time to perform MDS computation and o_s is the total overheads (including data transfer to and from GPU and thread scheduling), for s data points. Then, the total running time of the algorithm, T_s is given by:

$$T_s = c_s + o_s \tag{1}$$

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:

$$T'_s = 2{\cdot}c_s + o_s \tag{2}$$

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). We note that the overheads change with the input data size. 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.

4.3 Performance across different environments

We compared the performance of OpenCL MDS implementation in three environments (Table 1): (a) Tesla C1060, (b) Tesla "Fermi" M2070, and (c) EC2 cloud GPGPU Tesla "Fermi" M2050. We also analyzed the performance of double precision MDS in the two Tesla "Fermi" environments and compared with the single precision performance as presented in Figure 7. Single precision performance of MDS in Tesla "Fermi" M2070 was approximately 20% better than the EC2 cloud "Fermi" M2050. Tesla C1060 single precision performance was significantly lesser than both the "Fermi" GPGPUs. Both the "Fermi" GPGPUs performed similarly for the double precision MDS computations, with EC2 slightly outperforming "Fermi" M2070 at times. Double precision performance of "Fermi" M2070 for MDS was slightly above 1/3 of the single precision performance of "Fermi" M2070. The speedup of double precision OpenCL MDS over sequential MDS on CPU was better on EC2 than on bare metal "Fermi" nodes. This can be attributed to the fact that CPUs suffer from virtualization overheads in cloud computing environment, while GPUs do not, leading to better speedups of GPUs over CPUs in a cloud computing environment.

5 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 within the set. PageRank of a certain document depends on the number and the PageRank of other documents linked to it.

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

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], which creates a challenge in optimizing PageRank on GPUs. Efficient sparse matrix-vector multiplication is challenging on GPUs even for uniformly distributed sparse matrices [4].

There have been several efforts to improve the performance of PageRank calculations on GPUs, which require significant amount of pre-processing on the input sparse matrix [29, 30]. As noticed by the others, the storage format of sparse matrices plays a significant role in terms of the computation efficiency and the memory usage of the PageRank calculation.

5.1 Hybrid CSR-ELLPACK Storage-based PageRank

We first experimented by using a modified compressed sparse row (CSR) [4] format and a 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. Using CSR alone in a naïve manner results in poor performance due to imperfect load balancing of the rows. The ELLPACK format works best for uniformly distributed sparse matrices with similar length rows. Hence we implemented a hybrid approach where we used a preprocessing step to partition the rows 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 10 for the Stanford-Web data set [26]. More details about this experiment is given in subsection 5.3.3.

5.2 Balanced CSR Sparse Matrix Storage Format-based PageRank

In this implementation we propose a novel sparse matrix storage format optimized for irregularly distributed sparse matrix-vector multiplication computations on GPGPUs. This new format, which we call BSR (Balanced CSR), requires a fraction of more space than CSR and takes much lesser space than many other sparse matrix storage formats such as COO [4] or ELLPACK. BSR format can be constructed in a straightforward manner from CSR or other formats incurring only a small pre-processing overhead. This format is currently optimized for web-link matrices by eliminating the data array of the sparse web-link matrix, following the similar optimization to CSR and ELLPACK described in the previous section. However, the BSR format can be easily extended to support general purpose sparse matrices.

5.2.1 Constructing BSR Format Matrices

BSR format has a fixed width for the rows of the resultant dense matrix. It stores the row pointer followed by the column indices for the non-zero elements of the original matrix rows and lays the rows of input matrix continuously, even across row boundaries. This can result in partial rows of the original sparse matrix getting stored across multiple rows in the BSR dense matrix. BSR format and the corresponding PageRank algorithm supports the handling of partial rows. It uses three special characters: (a) α to mark the beginning of a new row in the original sparse matrix, (b) β to mark the beginning of a new partial row in the original matrix, and (c) ϵ to mark empty spaces. The construction of the new format uses the following steps, where A stands for the input sparse matrix and BSR stands for the output dense matrix.

```
foreach row R in A
if space_remaining(currentRow of BSR) <= 2
   ϵ fill bsrRow
   bsrRow ⇐ newRow(BSR)

if (freespace(bsrRow)+2) ≥ length(R)
   store α
   store Row pointer of R
   store the column pointers of R

else // length(R) > (freespace(bsrRow)+2)
   store β
   store Row pointer of R
   store column pointers of R
   while(moreColumnPointers in R)
      bsrRow = newRow(BSR)
      store β
      store Row pointer of R
      store column pointers of R
```

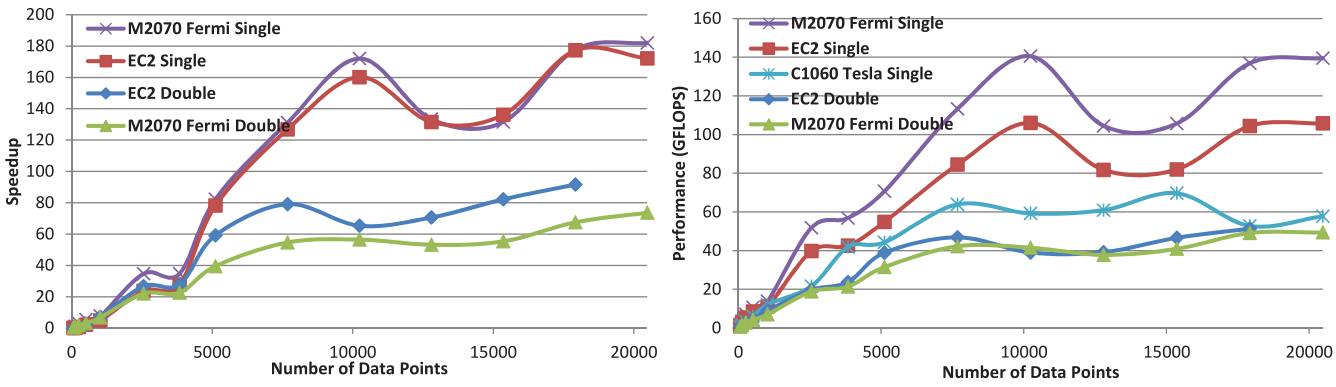


Figure 7 Performance comparison of MDS in different environments

$$A = \begin{bmatrix} 0\,1\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,0\,1\,1 \\ 0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0 \\ 0\,1\,1\,0\,1\,1\,0\,0\,0\,0\,0\,0\,0\,0\,1 \end{bmatrix} \qquad BSR = \begin{bmatrix} \alpha\,1 & 2 & 3\,\alpha & 2 & 1 & \epsilon \\ \beta\,3 & 1 & 2\,3 & 4 & 5 & 6 \\ \beta\,3 & 7 & 8\,9 & 10 & 11 & 13 \\ \beta\,3 & 14\,\alpha & 4 & 3 & 8 & \epsilon \\ \alpha\,5 & 2 & 3\,5 & 6 & 14 & \epsilon \end{bmatrix} \qquad P = \begin{bmatrix} \epsilon\ \epsilon\ \epsilon\ \epsilon \\ 3\,x\ \epsilon\ \epsilon \\ 3\,x\ \epsilon\ \epsilon \\ 3\,x\ \epsilon\ \epsilon \\ \epsilon\ \epsilon\ \epsilon\ \epsilon \end{bmatrix}$$

Figure 9 BSR example. A: input web link sparse matrix. BSR: output matrix. P: partial PageRank matrix template.

An example of this representation is shown in Figure 9.

Space overhead of this format over CSR is inversely proportional to the width of the BSR. However, using a wider matrix for BSR results in fewer rows, leading to underutilization of the GPU. At the same time the number of partial rows is inversely proportional to the width of BSR.

5.2.2 Comparison With COO Approach

The closest related work for the BSR approach is the COO kernel presented by Yang et al. [30], which they consider as one of the top performing kernels for power-law matrices. In their implementation, they divide the 3 arrays of COO in to equal parts to equally distribute the workload to warps. However, they do not consider the case of a row crossing the boundary of an interval. Each warp perform the multiplication of each element in the interval with the PageRank vector, and a shared memory reduction performed *sum reduction* as the next step in the warp. According to the authors, this shared memory reduction can lead to low-thread level parallelism in case an interval contains several rows of the input matrix. The BSR storage format we propose uses much less storage space than COO. Also their approach cannot handle partial rows in an interval, which would lead to load unbalances as some intervals can have less amount of work than others. Also they'll have to perform special preprocessing and padding of the COO matrix to ensure no rows will be crossing the interval boundaries. This would also make the length of the largest row the lower limit for interval size, which can result in very large intervals reducing the thread level parallelism as power-law matrices have few rows which are orders of magnitude longer than others. Our approach handles partial rows and ensures much better load balancing. Our approach requires *sum reductions* only for the partial rows and even in that case, the *sum reduction* is performed in a much higher granularity (per element in COO vs per partial row in our approach) in a highly parallel manner incuring very minimal overhead.

5.2.3 Sparse Matrix-Vector Multiplication Using BSR

Each row of the BSR matrix is processed by a single GPU thread. Fixed row widths result in evenly balanced load among the work items. Moreover, fixed row widths allow column-major storage, encouraging coalesced memory access by the threads in a work group and leading to significant performance improvement.

Table 2 Web data sets

Data Set	Nodes (Pages)	Edges (links)
BerkStan-Web	685,230	7,600,595
Google-Web	875,713	5,105,039
Stanford-Web	281,903	2,312,497

PageRank of each full row of input matrix A stored in a row of BSR will get calculated and stored by a single thread. However, with this format each row of the resultant matrix BSR can contain a maximum of two partial rows of A. Partial rows require aggregation across the worker items. We use the matrix P shown in Figure 9 to store the partial page ranks that result from the partial rows of A stored in the rows of BSR. P is an $N \times 4$ matrix, where N is the number of rows in BSR. Each row of P stores the page index and the values of the row-partial page rank for the corresponding row in BSR. We have a second kernel that performs the aggregation of these partial values to generate the final page rank value for these rows. This aggregation is peformed in parallel as shown in the PartialRankSum kernel in Figure 8. This *reduction* incurs only a small overhead compared to the overall computation. The overhead ranged from 1% to 2.5% of the total computation time for the data sets we considered. Our approach ensures high-thread level parallelism for all reasonable interval lengths.

There is room for improvement to optimize the random lookups of the PageRank vector by utilizing constant and texture memories. For any non-trivial web-graph the PageRank vector will be much larger than the size of constant and texture memories of the GPU. Hence, optimizing this would require more preprocessing to identify localities.

Performance of the BSR PageRank for the Stanford-Web data set compared against the above-mentioned hybrid version and CSR PageRank in CPU is shown in Figure 10. Figure 11 depicts the performance of BSR for three web data sets from Stanford large network data set [26] given in Table 2, and the speedup of BSR over single core CPU running BSR PageRank. According to Figure 11, BSR PageRank performs better with larger number of edges to number of pages ratios, specially on Fermi. Note that these results make it evident that BSR is not a suitable layout scheme for CPU execution.

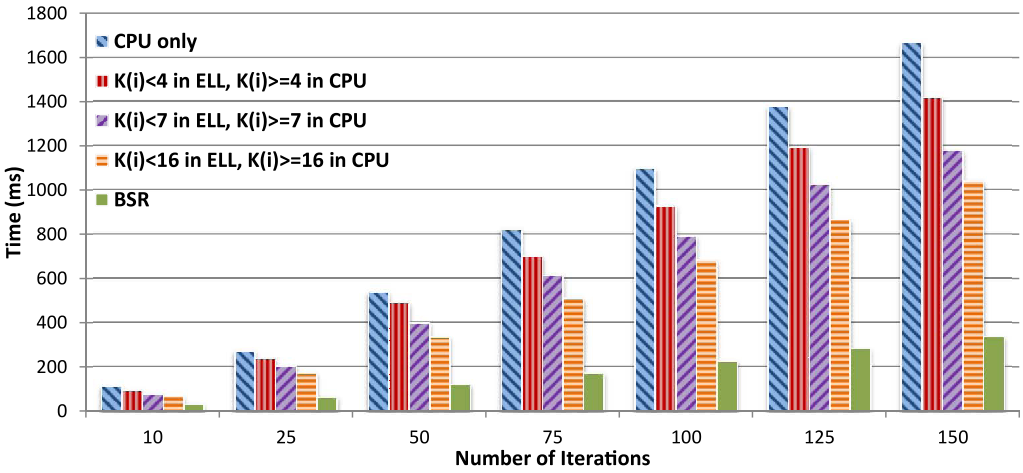


Figure 10 Performance of PageRank implementations for web-Stanford data set.

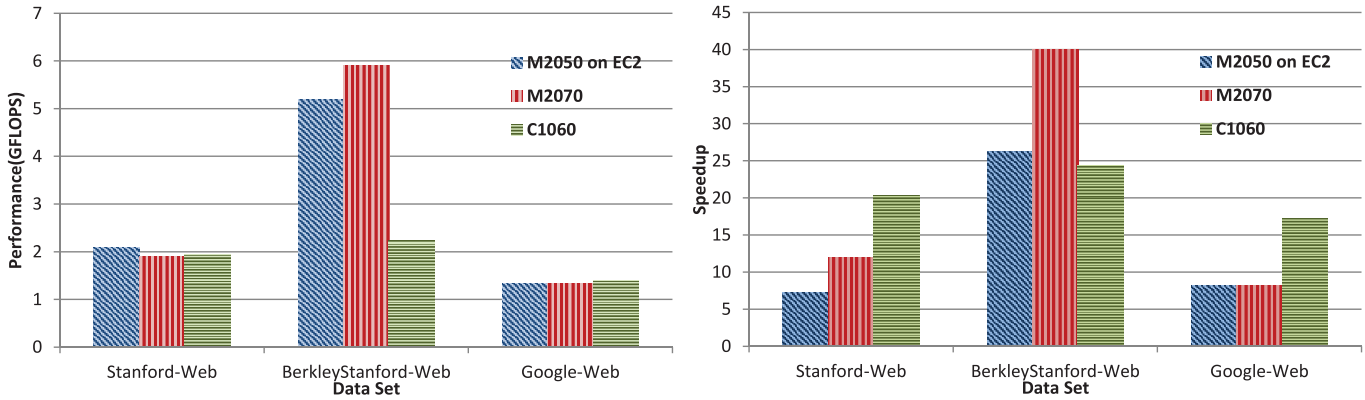


Figure 11 PageRank BSR implementation performance for different data sets. Speedups are over one CPU core.

5.3 Optimizations

5.3.1 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.

5.3.2 Optimizing Memory Access

We store the BSR matrix in column-major format enabling coalesced memory access from the threads in a warp. This resulted in significant speedups of the computation. The index array in the ELLPACK format is stored in appropriate order to enable contiguous memory accesses.

5.3.3 Sharing Work between CPU and GPU

Due to the power law distribution of non-zero elements, a small number of rows contain 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 10.

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 10 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

```
__kernel FixedWidthSparseRow(int* data,
          int* outLinks, float* ranks,
          float* newRanks, float* partials)
{
  blockNum = get_global_id(0);
  initPartials(gid, partials);
  newRank = 0;
  multiPage = 0;

  status = data[blockNum][0];
  page   = data[blockNum][1];
  for (int i=2; i < BSRROWWIDTH; i++){
    value = data[blockNum][i];

    // Value is an inLink
    if (value>=0) newRank += ranks[value];

    // Value is a row end filler
    else if (value==-3) break;

    // Beginning of a new page record
    else{
      Update(newRank, page, status, 0);
      i++;
      status = value;
      page = data[blockNum][i];
      newRank = 0;
      multiPage = 1;
    }
  }

  Update(newRank, page, status, (2*multiPage));
}

void Update(newRank, page, status, pIndex)
{
  newRank =(((1-D)/NUMPAGES) + (D*newRank));
  if (status == -2) {
    partials[blockNum][pIndex] = page;
    partials[blockNum][pIndex+1] = newRank;
  } else {
    newRanks[page] = newRank/outLinks[page];
  }
}

__kernel void PartialRankSum
        (float* newRanks, float* partials)
{
  blockNum = get_global_id(0);
  index = partials[blockNum][2];
  if (index > -1) {
    float rank = partials[blockNum][3];
    block++;
    while (partials[blockNum][0]== index){
      rank += partials[blockNum][1];
      block++;
    }
    newRanks[index] = rank;
  }
}
```

Figure 8 Outline of BSR based PageRank algorithm in OpenCL.

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.

We do not share work with CPU for the BSR computation.

5.4 Performance Across Different Environments

All three environments mentioned in Table 1 perform similarly for the Google-Web and the Stanford-Web data sets. For the BerkStan-Web dataset, the Fermi GPGPUs performed better than the Tesla GPGPU. The better performance of Fermi GPGPUs can be attributed to the hardware-managed caches that can improve the performance of random PageRank vector lookups.

6 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 [21].

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.

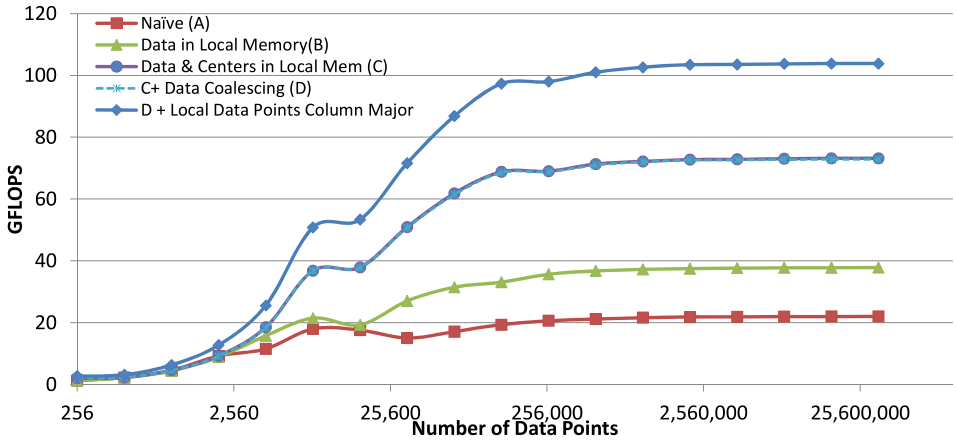


Figure 13 K-Means performance with the different optimizations steps, using 2D data points and 300 centroids.

```
__kernel KMeans(double* matrix,
  double* centroids,int* assignment, ){

 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)];
 }
 barrier(LOCAL MEM);

 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 12 Outline of K-Means in OpenCL.

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 12 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 13 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.

6.1 Caching Invariant Data

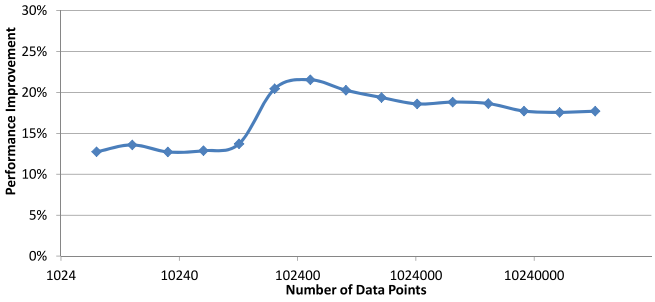
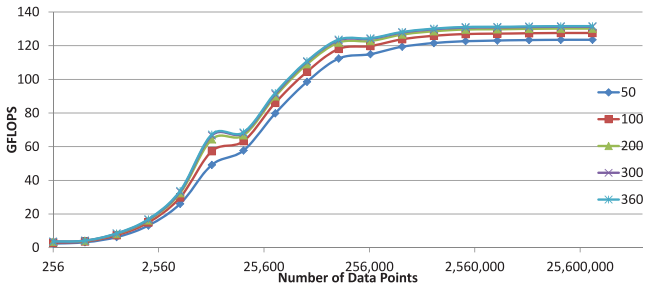
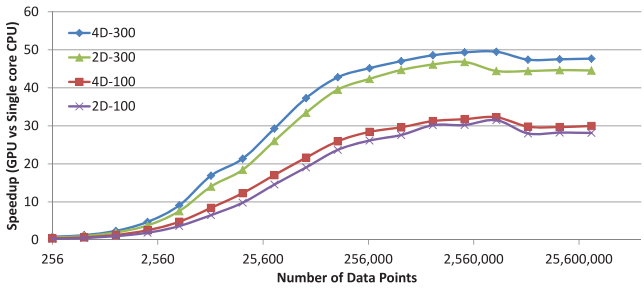


Figure 14 Performance improvement in K-Means due to caching of invariant data in GPU memory.

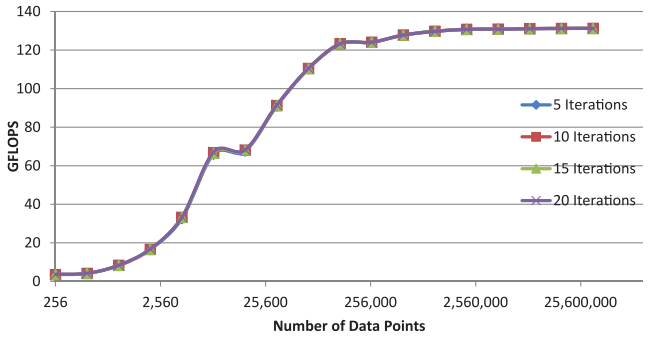
Transferring input data from CPU memory to GPU memory incurs major cost in performing data intensive statistical computations on GPUs. The speedup on GPU over CPU should be large enough to compensate for this initial data transfer cost. However, statistical iterative algorithms have loop-invariant data that could be reused across iterations. Figure 14 depicts the significant performance improvements gained by reusing of loop-invariant data in K-Means compared with no data reuse



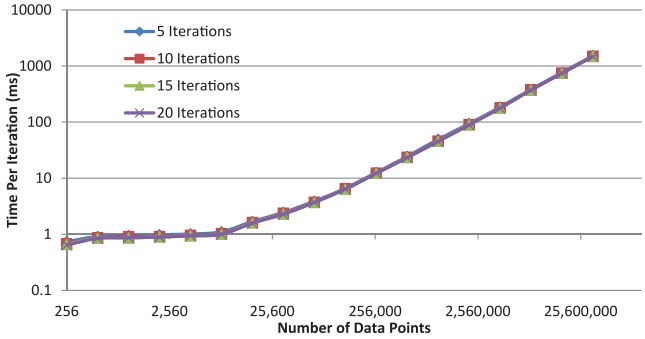
(a) K-Means: varying number of centers, using 4D data points.



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



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



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

Figure 15 K-Means with varying algorithmic parameters.

(copying the loop-invariant data from CPU to GPU in every iteration).

6.2 Optimizations

6.2.1 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 (on a 3 GHz Intel Core 2 Duo Xeon processor, with 4 MB L2 cache and 8 GB RAM).

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 in the Tesla c1060, as the next line, marked "B", shows. However, this resulted in a performance degradation in the newer Fermi architecture GPGPU's. We believe the L1 cache of Fermi architecture ensures this optimization.

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 13.

The performance curves changes at 8192 data point in Figure 13. We believe that this is due to the GPU getting saturated with threads at 8192 data points and above, since we spawn one thread for each data point. For data sizes smaller than 8192, the GPU kernel computation took a constant amount of time, indicating that GPU might have been underutilized for smaller data sizes. Finally, the flattening of the curve for large data sizes is likely because of reaching memory bandwidth limits.

6.2.2 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 13.

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 13. We believe that this is due to reduced bank conflicts when different threads in a local work group access local memory concurrently. Performing the same transformation for centroids in local memory did not result in any significant change to

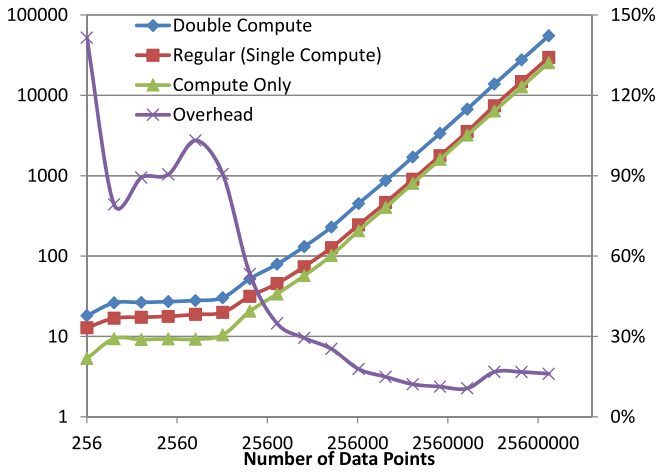


Figure 16 Overheads in OpenCL K-Means.

the performance (not shown in the figure). We believe this is because all the threads in a local work group access 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 15(a) presents the performance variation with different number of centroids, as the number of data points increases. Figure 15(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 15(c) and 15(d) show that there is no measurable change in performance with the number of iterations.

6.2.3 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 assignment 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.

6.3 Overhead Estimation

Following the model that was used for MDS in Section 4.2, we performed similar experiments for estimating kernel scheduling and data transfer overheads in K-Means Clustering. 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.

6.4 Performance across different environments

We compare the performance of OpenCL K-Means Clustering implementation in the three environments listed in Table 1. We also analyzed the performance of double precision MDS in the two Tesla "Fermi" environments and compared with the single precision performance as presented in Figure 17. Fermi and cloud-based GPGPU exhibited comparable performance. Interestingly, single-precision and double-precision in both the environments achieved similar (within 10% range) performance peaking around 100 GFLOPS. However, the speedups over sequential CPU computations were better on Amazon EC2 than on bare metal Fermi nodes. The most likely reason is that the virtualization overheads are high for CPU execution, but low for GPGPUs. This indicates that a good incentive exists to leverage cloud-based GPGPUs. One interesting observation is the C1060 performing better than newer Fermi architecture cards for the single precision computations. The reasons might be the overhead of ECC on Fermi and a possible suboptimal usage of L1 cache in Fermi. Unfortunately current NVIDIA OpenCL implementation do not provide a mechanism to disable or configure the size of the L1 cache.

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 four 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. Our findings can be summarized as follows:

1. Since parts of the algorithms tend to employ sparse data structures or irregular memory accesses it is useful to carry out portions of computation on the CPU.

2. In the context of clusters of GPUs, inter-node communication needs to go through CPU memory (as of the writing of this paper in mid-2011). This makes computing on the CPUs a compelling alternative on

data received from remote nodes, when the CPU-memory to device-memory data transfer times would more than offset any gains to be had running the algorithms on the GPUs.

3. Whenever possible, caching invariant data on GPU for use across kernel invocations significantly impacts performance.

4. While carefully optimizing the algorithms using specialized memory is important, as past studies have found, iterative statistical kernels cause complex trade-offs to arise due to irregular data access patterns (e.g., in use of texture memory) and size of invariant data (e.g., in use of constant memory).

5. Encoding algorithms directly in OpenCL turns out to be error-prone and difficult to debug. We believe that OpenCL might be better suited as a compilation target than a user programming environment.

In the rest of this section we elaborate on these findings.

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 other 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.

GPU caching of loop-invariant data. There turns out to be a significant amount of data that are invariant and used across multiple kernel calls. Such data can be cached in GPU memory to avoid repeated transfers from the CPU memory in each iteration. However, in order to harness this benefit, the loop-invariant data should fit in the GPU global memory and should be retained throughout the computation. When the size of loop-invariant data is larger than the available GPU global memory, it is more advantageous to distribute the work across compute nodes rather than swapping the data in and out of the GPU memory.

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. Interestingly, that is true even on the Fermi architecture even though it has a hardware managed cache. 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.

Leveraging Texture Memory. Texture memory provides a way to improved memory access of read-only data that has regular access pattern in a two-dimensional mapping

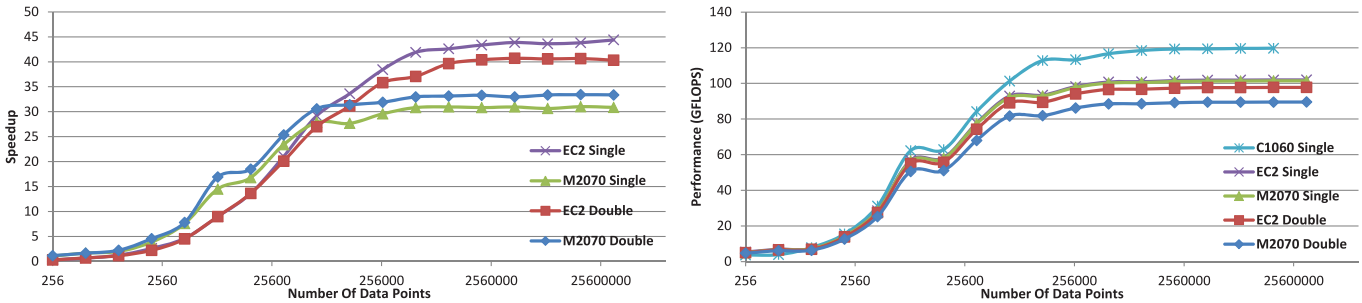


Figure 17 K-Means performance comparison across different environments.

of threads when the threads access contiguous chunks of a two-dimensional array. In the kernels we studied we found that the best performance was achieved when threads were mapped in one dimension, even when the array was two-dimensional. Each thread operated on an entire row of the array. As a result, our implementation was not conducive to utilizing texture memory.

Leveraging Constant Memory. As the name suggests, constant memory is useful for keeping the data that is invariant through the lifetime of a kernel. Unfortunately, the size of the constant memory was too small to keep the loop-invariant data, which do not change across kernel calls, for the kernels that we studied. However, since the data are required to be invariant only through one invocation of the kernel, it is possible to use constant memory to store data that might change across kernel calls as long as there is no change within one call of the kernel. The potential benefit comes when such data exhibit temporal locality, since the GPU has a hardware cache to store values read from constant memory so that hits in the cache are served much faster than misses. This gives us the possibility to use constant memory for the broadcasting of loop-variant data, which are relatively small and do not change within a single iteration. Still the loop-variant data for larger MDS and PageRank test cases were larger than the constant memory size.

Optimizing Data Layout. 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. In the case of PageRank, data layout has a dramatic impact on performance, since it leads to not only improved coalescing but also higher occupancy of the GPU cores, as an improved layout changes the way the work is divided across threads.

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.

Amazon EC2 GPU instances. On Amazon EC2 cloud computing environment, GPUs have lower virtualization overheads than CPUs. As a result, the GPU to CPU speedup is much higher than what is achieved on bare hardware. However, single and double precision performance are comparable.

8 Related Work

Emergence of accessible programming interfaces and industry standard languages has tremendously increased the interest in using GPUs for general purpose computing. CUDA, by NVIDIA, has been the most popular framework for this purpose [22]. In addition to directly studying application implementations in CUDA [12, 24, 31], there have been recent research projects exploring CUDA in hybrid CUDA/MPI environment [23], and using CUDA as a target in automatic translation [19, 18, 3].

There have been several past attempts at implementing the K-Means clustering algorithm on GPUs, mostly using CUDA or OpenGL [25, 13, 28, 20, 16]. Recently, Dhanasekaran et al. have used OpenCL to implement the K-Means algorithm [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 SMACOF implementation by Bae et al. [1]. Glimmer is another multilevel MDS implementation [14]. While Glimmer implements multilevel MDS using OpenGL Shading Language (GLSL) for large data sets, Bae used an interpolated approach for large data sizes, 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. Our MDS implementation uses the SMACOF iterative majorization algorithm. SMACOF is expected to give better quality results than Glimmer, even though Glimmer can process much larger data sets than SMACOF [14]. Since our study is in the context of GPU

clusters, with potentially vast amounts of distributed memory, we traded off in favor of a more accurate algorithm.

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 [29]. More recently, Yang et al. leveraged this property to auto-tune sparse matrix-vector multiplication on GPUs [30]. 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 implemented the SMACOF multi-dimensional scaling algorithm on the GPUs (the first GPU implementation, to the best of our knowledge) and a devised a novel data layout scheme for irregular sparse matrices suitable for GPU-based parallelization. In the case of PageRank, this turns out to have a significant impact on performance. We also implemented an optimised KMeansClustering algorithm. We evaluated three optimization techniques for each kernel, 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. In general, data layout is important in certain cases, and 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. We also showed that Amazon EC2 cloud GPU instances as a viable enviroment to perform comptuations using the above kernels.

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.

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.

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

[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.

[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.

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

[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.

[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.

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

[11] T. Gunarathne, B. Zhang, T.-L. Wu, and J. Qiu. Portable parallel programming on cloud and hpc: Scientific applications of twister4azure. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 97 –104, dec. 2011.

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

[13] 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.

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

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

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

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

[29] 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.

[30] 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.

[31] H. Zhou, K. Lange, and M. A. Suchard. Graphics processing units and high-dimensional optimization. *Statistical Science*, 25(3):311–324, 2010.