Accelerating Data Clustering on GPU-based Clusters under Shared Memory Abstraction

Konstantinos I. Karantasis, Eleftherios D. Polychronopoulos, George N. Dimitrakopoulos
High Performance Information Systems Lab
School of Computer Engineering and Informatics
University of Patras
Rio, Greece 26500
Email: {kik, edp, dimitrag}@hpclab.ceid.upatras.gr

Abstract—Many-core graphics processors are playing today an important role in the advancements of modern highly concurrent processors. Their ability to accelerate computation is being explored under several scientific fields. In the current paper we present the acceleration of a widely used data clustering algorithm, K-means, in the context of high performance GPU clusters. As opposed to most related implementation efforts that use MPI to port their target applications on a GPU cluster, our implementation follows the Software Distributed Shared Memory (SDSM) paradigm in order to distribute information and computation across the accelerator cluster. In order to investigate the efficiency of a programming model that offers shared memory abstraction on GPU clusters we present two implementations, one that is based on a SDSM implementation of OpenMP and another that utilizes the Pleiad cluster middleware on top of the Java platform. The first results show that such an implementation is feasible in order to accelerate a broad category of large scale, data intensive applications, among which K-means is a characteristic case.

I. INTRODUCTION

Data clustering constitutes an important procedure in many scientific and technological fields, including among others bioinformatics, data mining, economics research and social networks [1]. One of the most widely used algorithms for data clustering is the K-means algorithm [2]. K-means owes its popularity to the fact that, in many cases, its simplicity is combined with an acceptable quality of clustering. Because K-means is a partition-based algorithm, it presents a quite appropriate candidate for parallelization. Therefore, along the several variations of the algorithm that try to improve its results, there are many implementations that aim to speedup its execution time under a specific parallel architecture.

Concerning the currently available parallel systems we are experiencing a major technological shift that is often described as the multicore revolution [3]. Along the process of producing next generation processors that will encompass many computational cores [4], the modern graphics processors (GPUs) drive, at the moment, the technological advancements in terms of the number of cores that they deploy. At the same time, clusters remain the main architectural organization for high performance systems. Therefore, the need to combine the advantages of current trends in parallel architecture, has recently led to the deployment of the first high performance GPU clusters [5].

In order to exploit the potential parallelism that is offered at the hardware level by multicore clusters, a considerable effort has to be made at the software stack. Particularly in the case of GPU clusters, synergetic computation schemes have to be implemented in order to coordinate the execution between CPU and GPU threads. The first steps towards the development of sophisticated middleware, that will comprise both runtime libraries and compilers destined for these platforms, have been made through the porting and the evaluation of computationally intensive applications, were the optimization is done by hand. In that way, it is estimated that the necessary experience will be gained to drive the development of modern system software for multicores.

In the current paper, following the same practice, we present the implementation of an intensive data clustering algorithm in order to study the potential acceleration on that application category. In parallel, we attempt to gain important insight concerning attainable programming models that target GPU-based supercomputing clusters.

The rest of the paper is organized as follows. In section II we refer to the research efforts that relate to the presented implementation. Section III describes our parallel implementation of the K-means algorithm, as well as the software platforms that were used to realize that implementation on GPU clusters. In Section IV we provide the performance evaluation of the proposed scheme and finally in Section V we draw our conclusions and refer to our future work.

II. RELATED WORK

Since the introduction of modern GPUs and their respective development tools, CUDA [6] and OpenCL [7], several scientific applications have been ported on these many-core architectures [8]. In the context of data clustering, most research efforts that have analyzed the performance of K-means clustering, concentrate on the optimization of execution...
on a single GPU. In particular, emphasis is posed on the minimization of data transfers between host and device codes. The implementation of Zechner and Granitzer [9] achieves a maximum of 14x speedup to a fully SIMD optimized GPU implementation. The restrictions to this specific approach are posed by the amount of memory that is available on the device side. In that case, an under-utilization of the GPU cores may be observed.

Ren Wu et al. [10], investigated the execution on a single GPU with very large data sets that cannot fit into GPU’s onboard memory. They used K-means as an example and they showed that the GPU-accelerated version can still offer a great performance boost over a highly optimized CPU-only version running on 8 cores. The GPU-accelerated version is 6x to 12x faster. Compared to other approaches, GPU-accelerated implementations provide better raw performance and better cost to performance ratios.

Another approach is presented by GPUMiner [11]. This is a parallel data mining system that takes advantage of both aspects of GPU utilization, computation and visualization. GPUMiner uses CUDA to implement data mining algorithms and DirectX for visualization. Their bitmap K-means implementation efficiently utilizes GPU computational resources and minimizes the GPU - CPU data transfer. The computation time that is spent on the GPU represents more than the 98% of the total running time. Hong-tao et al. [12] in order to accelerate compute-intensive portions of traditional K-means, they offload both the data objects assignment and k-centroids recalculation on the GPU. Their results demonstrated that the speed of GPU-based K-means can reach as high as 40x of the CPU-based K-means.

Concerning the programming model, most codes that target GPU clusters tend to use a programming model that is based on message passing[13]. Particularly, in most cases, an implementation of MPI is used and certain efforts have tried to integrate GPU programming environments and MPI in a unified development platform[14].

Research efforts that are based on the distributed shared memory (DSM) paradigm, have not yet been thoroughly evaluated, although such an approach could offer a transparent extension to the memory hierarchy cluster-wide. Gelado et al. [15] have presented recently ADSM, moving towards the evaluation of DSM on GPU accelerators, however ADSM is currently concentrating on the communication between host and device memories and, for the moment, it is not meant to utilize GPUs on a distributed environment. On the other hand, Zippy[16], being based on Global Arrays, provides shared memory abstraction on CUDA enabled clusters. It succeeds to do so by exposing part of the data sharing mechanisms to the application programmer, and in that way bypasses the need to provide a memory consistency protocol. Finally, Strengert et al.[17] have presented CUDASA, which operates as a language extension to CUDA. CUDASA is supported by a source-to-source compiler, and in order to realize inter-node communication on a cluster it uses MPI calls.

III. IMPLEMENTATION ON ACCELERATOR CLUSTERS

In this section we describe the implementation steps that were taken and the programming platforms that were used in order to achieve the execution of the K-means algorithm at a cluster wide level.

A. CUDA

Targeting GPU clusters that are supplied with NVIDIA graphics processors, the CUDA [6] programming environment was used in order to implement the portion of the algorithm that executes on the GPU device side and to realize the communication between the code executing on the GPUs and the local CPU threads executing on the host side. CUDA, is a simple programming environment that includes a run-time library and a compiler driver. Currently, CUDA is the most advanced tool that can efficiently utilize the afforded data parallel resources of the NVIDIA graphics processors.

Much of the embrace of CUDA as a successful toolkit to program many-core GPUs has taken place due to the fact that it extends the C programming language. The application programmer actually writes C code, and the NVCC compiler driver bifurcates the code into two portions. One portion is delivered to the CPU - the so-called host side - mainly for the coordination of the computation, while the other portion, involving intensive computations, is delivered to the GPU - the so-called device side - that executes the code in parallel. The execution programming model of CUDA is characterized as simultaneous multithreading (SIMT) and presents some relevance with the SIMD category of parallel programming models on vector processors.

B. Intel Cluster OpenMP

The current practice concerning the utilization of GPU clusters uses, almost exclusively, use some type of message passing as a cluster wide programming model and in most cases that model is specifically an implementation of MPI [18]. In order to evaluate an alternative approach that would provide shared memory abstraction across the cluster, in the current implementation we have firstly integrated to our scheme a cluster-enabled OpenMP implementation. This specific implementation is provided as part of the Intel C/C++ compiler and it is based on the well-established software DSM Treadmarks [19]. Its main difference with regular OpenMP implementations for multicore systems in relation to the application programming interface, concerns the default data sharing policy. In Cluster OpenMP, data are not shared by default, but they have to be explicitly declared shared and have their memory allocated through special function calls.

C. Pleiad

The second alternative in terms of cluster middleware that is considered in the current paper is Pleiad [20]. Pleiad is a cluster middleware that is based on the Java platform and enables transparent multithreaded execution across the physically distributed nodes, such as the nodes of a GPU-based accelerator cluster. Data sharing in Pleiad is based on objects
Algorithm 1 K-means multi-GPU algorithm

Require: Object Set \( E = \{ e_1, \ldots, e_n \} \). Number of clusters \( k \)
Ensure: Set of cluster centroids \( C = \{ c_1, \ldots, c_k \} \)

procedure K-MEANS
\( C \leftarrow (\text{initialize centroids}) \)
repeat
for all \( obj \in E \) do
update object membership
\( \)GPU\nend for
for all \( obj \in E \) do
Check changes in membership and update cluster size
\( \)CPU\nend for
\( C \leftarrow (\text{new centroids via reduction}) \)
until \#Membership changes \( \geq \delta \)
end procedure

When the assignment part is completed, a global reduction of the newly computed clusters has to take place. Because the new cluster centers will have to be propagated across the cluster, in the current implementation we have chosen to perform that operation on the CPU side, since it does not exhibit a high degree of parallelism. Therefore the reduction computation is taking place across the OpenMP threads and the new centroids are propagated on every node at the start of the new iteration through the activation of the memory consistency mechanisms at the shared memory abstraction layer.

## IV. BENCHMARK EVALUATION

In this section we describe the benchmark setup and the experimental platform. Next, the performance of the execution models that have been considered is evaluated.

### A. Experimental Platform

The experimental evaluation of the current implementation took place on a 4-node GPU-based accelerator cluster. The 4 nodes of the cluster were externally connected with 2 NVIDIA Tesla 1U computing blades, establishing one connection per node, that supplied each node with 2 Tesla C1060 graphics processors. Thus, each node was able to utilize 60 stream multiprocessors and a total number of 480 cores at the GPU side. In total, this specific configuration resulted on a GPU cluster with an aggregate of 240 stream multiprocessors and 1920 cores.

In terms of software middleware platforms, the Intel Cluster OpenMP [22] implementation that is supplied with the the Intel C/C++ Compiler (icc) version 11.1 was used for the realization of a C-based SDSM and Pleiad was used for the evaluation of our scheme on top of the Java based cluster middleware. In every scheme, the NVIDIA CUDA software development environment was incorporated, in order to utilize the many-core GPUs of each cluster node.

<table>
<thead>
<tr>
<th>Type</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>4096KB (Cache)</td>
<td>4GB (Device Global)</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Gigabit Ethernet via Gigabit Switch</td>
<td>PCI-E x16</td>
</tr>
<tr>
<td>Cluster</td>
<td>Intel Cluster OpenMP (ICC 11.1)</td>
<td>Pleiad (Java 1.6.0_16)</td>
</tr>
<tr>
<td>Middleware</td>
<td>Pleiad (Java 1.6.0_16)</td>
<td>CUDA Driver API 2.2</td>
</tr>
</tbody>
</table>

TABLE I

The simple version of K-means is classifying a given data set, based on attributes/features, given a certain number of clusters a priori. The K-means algorithm works in iterations. At the start the algorithm selects \( K \) objects at random, one for each cluster, that are called centroids. On each iteration, K-means assigns each data object to its nearest centroid, according to some kind of metric, i.e. the euclidean distance metric. Subsequently, the algorithm finds the new centroids by computing the mean out of all the objects on every cluster. The algorithm terminates when the overall changes in the centroids fall below a certain threshold.

In order to utilize the several GPUs that are available at the extent of the cluster, our implementation follows a two-level hierarchical parallelization scheme (Alg. 1). At the first level, the partitioning of the object space takes place, and each group of objects is assigned for examination on each OpenMP thread that is created. Every such thread also corresponds to the manipulation of a GPU context that is going to run on the device side. Usually an OpenMP thread is spawned for each available GPU. The second level of parallelism spawning is taking place at the GPU level, where the GPU threads are organized into 2D blocks that in turn are structured on a 1D grid of thread blocks. Each thread on every block is responsible for the assignment of a certain data object on the appropriate cluster.
Specifically, both implementations relied exclusively upon the low-level CUDA Driver API in order to utilize the NVIDIA GPUs. That decision was driven mainly by the fact that the use of the CUDA runtime environment was not possible in the case of Intel Cluster OpenMP because the Intel C/C++ compiler currently is not fully supported by the NVCC CUDA compiler driver. Particularly, the cooperation is feasible if the source code that is destined to run on the host is compiled with \texttt{icc} and the source code of the CUDA kernels that will operate on the GPU device side is compiled with NVIDIA CUDA compiler driver (\texttt{nvcc}). Currently the only restriction, under that cooperation scheme, is that no direct memory copy (\texttt{memcpy}) operation can take part between GPU and a memory area that is sharable cluster-wide through the Intel SDSM.

### B. Performance evaluation on GPU clusters

In order to evaluate the performance of our implementation we have conducted experiments using various large input files. The contents of these files represent real data sets that are supplied by the UCI Knowledge Discovery in Databases repository \cite{UCI}. In the following graphs we present the results that concern executions of K-means with input sizes of 1.6 million objects. On every run a number of 20 clusters was requested and these clusters were permanently determined after 106 iterations. The execution times refer to the average execution time of 5 runs.

Three basic configurations are compared in terms of execution time (Table II and Fig. 1) and their respective speedup (Fig. 2). \texttt{CUDA-LOCAL} refers to local multi-GPU evaluation on a single node of the cluster. This implementation is solely based on CUDA and does not involve interactions with any cluster middleware. \texttt{CUDA-SDSM} refers to a cluster wide evaluation over SDSM through the Intel Cluster OpenMP implementation. Under that scheme, a single process is started on every node, and in the case of utilization of 2 or more GPUs per node, each process uses internal local multithreading. Specifically, on the available cluster, the hardware resources signify the creation of maximum 2 threads per process. Finally, \texttt{CUDA-PLEIAD} corresponds to the evaluation of K-means through the Pleiad cluster middleware. In that case, Pleiad uses home lazy release consistency (HLRC) as a consistency protocol, and invalidations to propagate object changes. Also in Pleiad, internal multithreading is exercised transparently on every node.

<table>
<thead>
<tr>
<th>CPU/GPU Contexts</th>
<th>OpenMP CPU LOCAL</th>
<th>CUDA LOCAL</th>
<th>CUDA SDSM</th>
<th>CUDA PLEIAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>224.58</td>
<td>41.50</td>
<td>41.61</td>
<td>41.60</td>
</tr>
<tr>
<td>2</td>
<td>113.22</td>
<td>19.91</td>
<td>23.52</td>
<td>22.10</td>
</tr>
<tr>
<td>4</td>
<td>57.99</td>
<td>23.78</td>
<td>12.57</td>
<td>11.51</td>
</tr>
<tr>
<td>6</td>
<td>62.33</td>
<td>22.00</td>
<td>10.44</td>
<td>9.10</td>
</tr>
<tr>
<td>8</td>
<td>58.76</td>
<td>20.78</td>
<td>8.73</td>
<td>7.13</td>
</tr>
</tbody>
</table>

Speedups are presented in comparison to the sequential version of the K-means algorithm that is available through the NU-Minebench benchmark suite \cite{Minebench}. According to the results, the highest speedup, approximately between 56x and 11x, is observed under the evaluation with Intel Cluster OpenMP, while a positive speedup scaling is observed using Pleiad as well, with a maximum of 32x. In the case of \texttt{CUDA-LOCAL} the speedup does not continue to scale when a GPU is shared between 2 or more GPU contexts. Of course the same applies for the multicore execution on the CPU with \texttt{OpenMP} from 4 threads and beyond. This is an expected behavior due to the lack of hardware resources, and the results were included here for the sake of completeness.

A detailed presentation of the time spent on the various phases of an iteration is shown in Fig. 3. These results show that the impact of the acceleration of the code portion that is executed on the GPU is higher in comparison to the speedup of the reduction phase on the CPU. That effect is expected to be higher in the presence of larger data sets, if a successful memory mapping of these data becomes feasible on the device side. At the cluster level, the communication cost is lower under Intel SDSM. In Pleiad the communication cost is higher in comparison to Intel SDSM, mainly due to the inevitable serialization process on the shared objects that is taking place on the Java platform. However that cost does not surpass the overall benefit of GPU acceleration across the cluster.
V. CONCLUSIONS AND FUTURE WORK

In the current paper we have presented the implementation of K-means on high performance GPU clusters. The implementation followed a novel approach in terms of GPU cluster utilization that poses an interesting alternative to mainstream message passing. The multi-GPU computation on the cluster was carried out through the integration of our scheme with two cluster middleware platforms that offer multithreading under shared memory abstraction, Intel Cluster OpenMP and Pleiad.

First results from the experimental evaluation show that, for specific applications, the presented approach is valid and can result in a considerable acceleration of such applications. The proposed implementation, if compared with message passing, extends in a more consistent and transparent way the memory hierarchy of a GPU cluster. Therefore, these preliminary results can encourage the implementation of appropriate middleware that will increase programming productivity providing shared memory abstraction and concurrently will offer an efficient execution model for heterogeneous GPU clusters.

Our future research will concentrate on providing such infrastructure at the middleware level. In parallel, we will aim to evaluate our scheme with more applications that pose high demands in terms of both data handling and computational cost.

REFERENCES