Accelerating Text Mining Workloads in a MapReduce-based Distributed GPU Environment

Peter Wittek*, Sándor Darányi*

Swedish School of Library and Information Science, University of Borås, Borås, Sweden

Abstract

Scientific computations have been using GPU-enabled computers successfully, often relying on distributed nodes to overcome the limitations of device memory. Only a handful of text mining applications benefit from such infrastructure. Since the initial steps of text mining are typically data-intensive, and the ease of deployment of algorithms is an important factor in developing advanced applications, we introduce a flexible, distributed, MapReduce-based text mining workflow that performs I/O-bound operations on CPUs with industry-standard tools and then runs compute-bound operations on GPUs which are optimized to ensure coalesced memory access and effective use of shared memory. We have performed extensive tests of our algorithms on a cluster of eight nodes with two NVidia Tesla M2050 attached to each, and we achieve considerable speedups for random projection and self-organizing maps.

Keywords: GPU Computing, MapReduce, Text Mining, Self-Organizing Maps, Random Projection

1. Introduction

Graphics processing units (GPUs) were originally designed to accelerate computer graphics through massive on-chip parallelism. The inherent data-parallelism in graphics applications is also apparent in many other fields, and GPUs have evolved into powerful tools for more general cases of computationally intensive tasks. For instance, researchers have studied how GPUs

*Corresponding address: Swedish School of Library and Information Science, University of Borås, Allegatan 1, Borås, S-501 90, Sweden

can be applied to problem domains such as scientific computing [36, 45, 51] and visual applications [12, 42, 46].

Recent improvements in the programmability of GPUs have made graphics hardware an even more compelling platform for computationally demanding tasks. Compute Unified Device Architecture (CUDA) is a C-like language allowing the implementation of innovative data-parallel algorithms in general computing terms to solve many non-graphics applications such as database searching and sorting, medical imaging, protein folding, and fluid dynamics simulation. The disadvantage of CUDA is that it is primarily supported on NVidia devices, although cross-compilers exist for other architectures. Open Computing Language (OpenCL), which uses both task-based and data-based parallelism, is another framework designed for GPU programming supported by several hardware and software vendors, e.g. Apple, NVidia, Intel and AMD. Many CUDA based algorithms can now be ported to OpenCL, but OpenCL is still less popular than CUDA.

Nevertheless, the process of developing efficient GPU implementations is still highly application dependent, and many applications require significant re-structuring of the algorithms to realize high performance on GPUs. Given their extremely high computing demands, we believe that text mining applications provide a very interesting potential application domain for GPUs. There have been several examples of research in this direction, including inverted indexing [52], topic modelling [7, 9, 26, 31, 57], and document clustering [56, 59].

However, due to the limited size of device memory, many GPU based algorithms have low capability in solving problems with large datasets. Large-scale problems in HPC are normally solved on CPU clusters relying on powerful toolsets and APIs for multiple CPUs. GPUs have made a remarkable progress in HPC clusters. The Top500 list of supercomputers had eleven GPU-accelerated systems in November 2010, 37 in November 2011, and 54 in June 2012 (Nvidia and ATI/AMD combined). Positions close to the very top are being claimed by such systems, for instance, the Titan Supercomputer currently being built at the Oak Ridge National Laboratory with 15,000 GPUs is predicted to claim the top spot [50]. The high energy efficient is also worth noting: the Barcelona Supercomputing Center is building an experimental cluster of GPU-accelerated cell phone components that might point to a solution with regard to the energy barrier in exascale computing [39].

One popular toolset in distributed computing is the large-dataset processing model MapReduce [13], a high-level framework that puts the focus

on the problem and not the low-level details inherent in distributed applications, such as communication and resource allocation. MapReduce targets data-intensive problems, it also performs well on other large-scale tasks. This has lead to the development of GPU-based MapReduce models, which makes it easier to adapt algorithms to run on a distributed GPU cluster. A crucial advantage of MapReduce is that it also helps with the integration of other tasks, which in turn aids the development of advanced services, for instance, in digital libraries [43].

The key contribution of this paper is two-fold. Firstly, we introduce a MapReduce-based framework for text mining workflows on heterogeneous clusters, where I/O-bound operations are run on the CPU, and compute-intensive tasks are performed on the GPU. Secondly, we adapted two well-known text mining tasks, dimensionality reduction by random projection and visualization by self-organizing maps, to GPU-aware MapReduce jobs.

The organization of this paper is as follows. The computational power, energy and cost efficiency of GPUs have attracted considerable interest in the information retrieval, text and data mining community; we briefly overview the results we find the most relevant to our paper in Section 2. Distributed examples are rare, however, and the heterogeneity, the presence of both CPUs and GPUs is often understated. We discuss the MapReduce framework in Section 3, which is a tremendous help in a distributed setting. Then we introduce our flexible, MapReduce-based, GPU-aware framework through an example pipeline in Section 4. We present put experimental results in Section 5, and finally draw our conclusions in Section 6.

2. Text mining on GPUs

The initial step of text mining translates the unstructured documents to algorithmically more manageable representation, typically in the form of an inverted index or a similar structure. An inverted index consists of inverted lists, where each inverted list contains the document IDs of all documents in the collection that contain the word corresponding to the list. Each inverted list is typically sorted by document IDs, and usually also contains for each document the number of occurrences of the word in that document. This representation can be viewed as a row-major sparse matrix where the rows correspond to terms and the columns to documents (that is, a term-document matrix). Inverted indexing starts by parsing each document into a bag of words. Parsing consists of a sequence of simple processing steps:

tokenization, stemming, and removal of stop words. Tokenization splits a document into individual tokens at word delimiters, stemming reduces different forms of a term into a single common root, and the removal of stop words eliminates common terms, such as "the", "to", "and", etc., which do not carry meaning. A recent algorithm employs both CPUs and GPUs in a distributed fashion to achieve a throughput of 262 MB/s on the ClueWeb09 dataset [52]. The strategy includes a pipelined workflow that produces parallel parsed streams that are consumed at the same rate by parallel indexers; a hybrid trie and B-tree dictionary data structure in which the trie is represented by a table for fast look-up and each B-tree node contains string caches (this is particularly useful for indexing on the GPU); the allocation of parsed streams with frequent terms to CPU threads and the rest to GPU threads so as to match the throughput of parsed streams since the different hardware excels at different workloads; and an optimized CUDA indexer implementation that ensures coalesced memory access and effective use of shared memory. Recognizing the complexity of an efficient, GPU-assisted inverted indexing, [15] found that the implementation of a complete search engine on a GPU does not appear to be realistic, but accelerating query processing is a viable option. Focusing on maximizing throughput on a single machine, the authors worked on decompressing inverted indexes, Boolean set operations, and finding the top-k matching documents for a query.

Topic modelling is a good candidate for GPU acceleration. Latent semantic analysis (LSA), or latent semantic indexing [14], is similar to the generalized vector space model [55], it measures semantic information through co-occurrence analysis in the corpus. LSA treats the entire document as the context of a word being analyzed. In LSA the dimension of the vector space is reduced by singular value decomposition (SVD), which is a computationally demanding operation. By a bidiagonalization method, a speedup up to 8x has been achieved for large matrices [26]. The baseline CPU implementation used Basic Linear Algebra Subprograms (BLAS) calls of the highly optimized Intel Math Kernel Library on an Intel Dual Core 2.66GHz PC, whereas the GPU version relied on Compute Unified BLAS (CUBLAS) running on an NVIDIA GTX 280. Translating the problem at hand to BLAS operations is a very general approach and is an easy way to achieve high speedups, as BLAS calls are already available and optimized for a number of platforms, including both CPUs and GPUs. A similar effort developed an implementation six times faster than the CPU version, using the same graphics hardware and a slightly different translation of the original SVD

problem to BLAS calls [9].

Supervised semantic indexing (SSI) is another topic modelling approach used to rank documents in a corpus based on their semantic similarity to a given text query [2]. SSI searches for a direct association between the words contained in a document and its similarity score, also considering correlations between words due to synonymy and polysemy. The similarity score is based on the inner product of vectors in a reduced dimensional space, just like in LSA, but the model is trained in a way similar to an artificial neural network (ANN). The input nodes are the values of the term-document matrix, each connected directly to all output nodes that represent a predefined number of conceptual categories to be learned. The ANN can be represented as a bias vector b with a length of k and a weight matrix U sized $w \times k$, where w is the number of words in the vocabulary and k is the number of conceptual categories. The model is refined in subsequent iterations until a convergence condition is met. To efficiently parallelize the algorithm on GPUs, parallelism must be exploited across iterations [7]. Due to data dependency, the results are not entirely accurate, but sufficient for the purpose of SSI. This strategy is called dependency relaxation.

Latent Dirichlet allocation (LDA) is a generative probabilistic topic model of a corpus. The basic idea is that documents are represented as random mixtures over latent topics, where each topic is characterized by a distribution over words. LDA assumes the following generative process for each document d in a corpus D [4]. First, choose $\theta \sim \mathrm{Dir}(\alpha)$. Then, for each of the w terms t_n, choose a topic $z_n \sim \mathrm{Multinomial}(\theta)$, and choose a term t_n from $p(t_n|z_n, \beta)$, a multinomial probability conditioned on the topic z_n. Given this generative process, the task is to compute the posterior distribution of the hidden variables given a document: $p(\theta, z|d, \alpha, \beta) = \frac{p(\theta,z,d|\alpha,\beta)}{p(d|\alpha,\beta)}$. While the above formula is computationally intractable in most cases, approximations exist. Using collapsed Gibbs sampling with randomization, 26x speedup has been achieved over a serial implementation [57], and 196x speedup by using collapsed variational Bayesian [31, 57], which is a deterministic method that avoids randomization.

Moving beyond topic modelling, document clustering has also benefited from GPU acceleration. Document clustering is a sub-field of data clustering where a collection of documents is categorized into different subsets with respect to document similarity. K-means clustering partitions the data set into k subsets by assigning each point to the subset whose centre is the closest centre to the point. Then it recalculates the k cluster centres as

the geometric centres of the subsets. The algorithm repeats these two steps until no data point moves from one cluster to another. Using an Nvidia GeForce GTX 280, 11x speedup has been achieved over an optimized eight-core CPU version [56]. A recent flocking-based algorithm implements the clustering process through the simulation of mixed-species birds in nature. Each document is represented as a point in a two-dimensional space. The initial positions of documents are set at random. One point interacts with its neighbours according to a clustering criterion, for instance, a similarity metric between documents. The algorithmic complexity is inherently quadratic in the number of documents. A single GPU card over a single-node desktop for several thousand documents has achieved a speedup of up to five times [11], and a small four-node cluster with higher-end GPUs shows 30x-50x speedups [59].

Taking related work into consideration, we believe that the initial step of text mining workflow, inverted indexing should be executed on CPUs, preferably in a distributed fashion. We rely on random projection for topic modelling, which stands out among other approaches for its efficiency in using computing power, yet it can still benefit from distributed GPU programming. We add a visual clustering method as the final step of our workflow, self-organizing maps, which are known for their enormous computational needs.

3. Distributed computing in a heterogeneous environment

Heterogeneous computing aims to combine the parallelism of traditional multicore CPUs and GPU accelerator cores to deliver unprecedented levels of performance [5]. While the phrase typically refers to a single node, a distributed environment may be constructed from such heterogeneous nodes.

CPUs excel in running single-threaded processes, or in multithreaded applications in which a thread often consists of fairly complicated sequential code. Graphics processors are ideally suited for computations that can be run on numerous data elements simultaneously in parallel. This typically involves arithmetic on large data sets (such as matrices) where the same operation can be performed across thousands of elements at the same time. This is actually a requirement for good performance: the software must use a large number of threads. The overhead of creating new threads is minimal compared to CPUs that typically take thousands of clock cycles to generate and schedule, and a low number of threads will not perform well on GPU [21]. The decomposition and scheduling of computation among CPU cores and

GPUs are not trivial even on a single node [18, 27, 30], and the task is even more complicated for clusters [37]. In order to issue work to several GPUs concurrently, a program needs the same number of CPU threads, each with its own context. All inter-GPU communication takes place via host nodes. Threads can be lightweight (pthreads, OpenMP, etc. [25]) or heavyweight (MPI [24]). Any CPU multi-threading or message-passing API or library can be used, as CPU thread management is completely orthogonal to GPGPU programming. For example, one can add GPU processing to an existing MPI application by porting the compute-intensive portions of the code without changing the communication structure [1]. However, the efficient utilisation of all CPU and GPU cores remains an open question.

High-performance computing (HPC) uses supercomputers and computer clusters to solve advanced computational problems. A supercomputer is purpose-built hardware which is typically very costly to build. Clusters combine powerful workstations or even commodity hardware through a high-speed network to achieve higher scales. Since even commodity hardware is extremely powerful these days, enormous clusters have been overtaking supercomputers in the rankings of computational performance for the past decade.

The computational expense to execute data or text mining based analysis as advanced services in real-world applications such as digital libraries has been identified as the major cause for the lack of more widespread use of such services [43]. For the easy deployment of text mining applications, a higher level of abstraction is needed than the ones commonly used in HPC. MapReduce is a good candidate [13], already used in many text-related tasks such as inverted indexing [29]. MapReduce is not a novel framework in distributed computing, drawing on well-known principles in parallel and distributed computing, and assembling them in a way to scale to collections of sizes unseen before. Like OpenMP and MPI, MapReduce provides an abstraction, a means to distribute computation without burdening the programmer with the details of distributed computing; however, the level of granularity is different. MapReduce has been typically restricted to data-intensive processing, whereas HPC is compute-bound. More recently, however, there has been a few attempts to use MapReduce for HPC. Initial investigations have been carried out to develop an efficient MapReduce framework on heterogeneous, GPU-equipped nodes [44, 47].

In what follows, we explain some details of data-intensive MapReduce (Section 3.1). Then we discuss some options that are available to exploit the

framework in compute-intense tasks typical to GPU workloads (Section 3.2).

3.1. Data-intensive MapReduce

MapReduce is inspired by map and reduce functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as in their original forms [13]. The framework was originally developed for I/O-bound operations and data-intensive processing.

In the map step, the so-called master node (a coordinating computer in the cluster) takes the input, divides it up into smaller sub-problems, and distributes those to worker nodes. In turn, a worker node may do this again, leading to a multi-level tree structure: it processes that smaller problem, and passes the answer back to its master node.

The basic data structure of the framework is key-value pairs. Keys and values can be arbitrarily complex data structures. For instance, for a collection of web pages, the keys can be the URLs and the values are the HTML content. For a graph, keys can be the node identifiers, while values are the adjacency lists of those nodes. The output of the mapper is a sorted list of key-value pairs. The mapper also commonly performs input format parsing, projection (selecting the relevant fields), and filtering (removing records that are not of interest).

In the reduce step, the master node then takes the answers to all the sub-problems and combines them in a way to get the output - the answer to the problem it was originally trying to solve. As the processing gets more complex, this complexity is generally manifested by having more MapReduce jobs, rather than having more complex map and reduce functions. In other words, a developer thinks about adding more jobs, rather than increasing the complexity of the jobs. The two steps, map and reduce, are juxtaposed by an intermediate step, sort and shuffle. The sort phase orders the key-value pairs by a similarity function on the keys (note that keys can be arbitrarily complex structures), and the shuffle phase transfers the map outputs to the reducers as inputs.

Once a problem is broken down into map and reduce operations, then multiple map operations and multiple reduce operations can be distributed to run on different nodes simultaneously.

Perhaps the most notable implementation is Hadoop. Hadoop in fact is more than just a MapReduce framework, resulting from the tight integration of a distributed filesystem (HDFS) and a MapReduce framework (Hadoop

MapReduce). It is the earliest fully functional open source implementation of the original MapReduce paper published by Google [13]. The presence of a distributed filesystem in the package is not accidental. Hadoop was conceived with hundreds of individual nodes in mind and real scalability shows only beyond a certain number of nodes. To support this goal, the development of a reliable distributed filesystem was a must for the project to be successful. This alone would distinguish Hadoop from the rest of the frameworks overviewed in this manuscript: the other systems do not come with a distributed filesystem, and even if they do operate in a distributed fashion, the speed-up declines sharply after reaching 30-60 processing cores, which translates to roughly 8-30 nodes, depending on the configuration of individual nodes. Hadoop MapReduce launches redundant jobs, the same chunk of data is distributed to at least three nodes. A node may run multiple jobs simultaneously, hence exploiting internal parallelism of multicore nodes. To do so, Hadoop launches multiple Java virtual machines on the node. While a solid solution, the computing efficiency of such an architecture is dubious. This is the reason why Hadoop emphasizes its focus on data-intensive computing, that is, a large volume of data has to be processed reliably, and the execution time is of secondary importance. Being the most widespread framework, a variety of algorithms are readily available for Hadoop. Mahout offers a limited number of machine learning libraries [35].

A venerable MapReduce framework that uses the parallelism of multicore processors, but does not scale to multiple computers is Phoenix++ [49]. The great advantage of Phoenix++ is that the code base has been rewritten for the third time and it is both quite mature [40, 58] and extremely educational to develop MapReduce jobs with it. Since Phoenix++ is not distributed, debugging is much simpler. Once a MapReduce job takes shape and works with Phoenix++, it can be adapted to other frameworks. Phoenix++ demonstrates extremely well that the overhead of MapReduce might be quite minimal. The authors compared it to other shared-memory parallel programming libraries such as Pthreads and OpenMP, and found Phoenix++ performing just as well. Phoenix++ is well-suited for compute-intensive tasks which can be easily formulated in pure MapReduce jobs.

Another implementation, MR-MPI[38], was originally targeted at compute-bound biomedical applications. It is similar to Phoenix++ in that it uses only the CPUs. However, it uses MPI to exploit parallelism, and that also means that it scales to multiple nodes. It is not a pure framework, it allows certain types of communications via MPI which violate the strict MapReduce

principles.

3.2. GPU-aware MapReduce

GPUs excel at compute-bound operations, but it is not always easy to formulate a problem in the data parallel fashion that is required by lower level frameworks to program GPUs. There are several research attempts that port MapReduce to GPUs to help develop applications faster on this kind of hardware.

The core concept of GPGPU computing is the kernel. A kernel is a unit of code that will execute on a graphics device launched from the host CPU. The execution is asynchronous in most cases: the host may continue working on other tasks while the kernel is being executed. The graphics device cannot access the main memory, the data has to be copied to the memory of the device through the system bus. This is a costly operation, and if a sufficient level of parallelism cannot be achieved in the kernel, GPU-based computation may decrease the overall performance due to this overhead [1].

GPUs are made of a large number of relatively simple processing cores (also called streaming processors or CUDA cores). The cores are organized in groups, for instance, in NVidia GPUs, eight streaming processors make a streaming multiprocessor (SMP). Each streaming processor has a very limited memory in the form of registers, and an SMP shares a small amount of local memory (also called shared memory). The access to the device's main memory (also called global memory) is expensive, and it should be avoided by the programmer by carefully using the shared memory of SMPs. Access to the shared memory is fast, as long as threads do not attempt to access the same location (bank) simultaneously (also known as a bank conflict). The scheduler of SMPs attempts to reduce delays caused by global memory accesses by replacing the running threads with other available threads until the memory fetch is finished. This can only be done if a sufficiently large number of threads are available. Apart from stating a problem as a data-parallel algorithm, it is essential to optimize shared memory usage and always provided a sufficient load to efficiently use GPU resources.

Mars was the first framework to program a GPU with the MapReduce paradigm [17]. While it did show good potential compared to a CPU-based MapReduce, it does not utilize the GPU efficiently. It has not been updated for a long time and it is not capable of using more than one GPU.

While outdated and somewhat inefficient, it does show an important difficulty of GPU-based text processing: GPUs prefer keys of a fixed size, and

tokens in a text stream never come in a fixed size. To overcome this problem, Mars employed a hash function and provides an example of a word counting job.

Inspired by Mars, GPMR is the latest attempt to harness the power of GPUs with MapReduce [47]. It can use any number of GPUs in a node and it is also capable of running in a distributed system. Both features are provided by MPI. The performance efficiency declines after about 16 GPUs in most of the tests. GPMR does not try to hide the complexity of GPU programming, and it allows full control over the individual GPUs. It is a compromise made to achieve maximum performance. The default MapReduce scheduler moves the data from the GPU to the main memory after each map step, and then pushes it back before reduction (if needed), which might be inefficient in certain applications.

Taking the approach of GPMR further, the MapReduce and GPU parts of an algorithm can be entirely decomposed. This way, for instance, one can use MR-MPI or Hadoop for a high-level load distribution and use GPU code only inside the map or reduce jobs. This is the approach we have taken.

4. Adapting a text mining workflow

In what follows, we explain a distributed text mining workflow. As the initial step, it uses CPU-based MapReduce on a cluster for the I/O-bound operation of inverted indexing, which might also include steps that are difficult to reimplement on the GPU (Section 4.1). With the availability of GPU-aware MapReduce frameworks, we are able to efficiently leverage on GPU-enabled nodes for topic modelling (Section 4.2) and visual clustering (Section 4.3).

4.1. Building an inverted index

Text processing is a tedious, error-prone procedure which has to take a wide array of issues into consideration, including font encodings, punctuation, language-specific problems such as finding word limits in Chinese or stemming in most European languages, etc. Moreover, this initial step of text mining is I/O-bound, it will always be limited by the hard disk access times and transfer rates. Therefore we believe it is a more sensible approach to leverage on proved technologies to build the first element of a workflow.

Lucene is the industry-standard information retrieval software library that builds an inverted index [16], which can be interpreted as a row-major

sparse representation of a term-document matrix. The indexing engine is heavily optimized, and is able to include the usual pre-processing steps, including stemming in a variety of languages. The core engine of Lucene is independent of the file format. Text from PDFs, HTML, Microsoft Word, and OpenDocument documents, as well as many others can all be indexed as long as their textual information can be extracted. A range of language-specific tools is also provided.

Nutch is an open source web-search project that relies on Hadoop to distribute the workload of crawling and indexing, and uses Lucene as its back-end for building the inverted index [8]. Nutch is particularly well suited for scaling out with a large number of commodity hardware [32, 33].

4.2. Random projection

Random projection does not rely on the use of computationally intensive matrix decomposition algorithms like singular value decomposition. This makes random projection a much more scalable technique in practice. Instead of first constructing a huge co-occurrence matrix and then use a separate dimension reduction phase, random projection builds an incremental word space model [20, 41]. The random projection technique can be described as a two-step operation:

- First, each context (e.g. each document or each word) in the data is assigned a unique and randomly generated representation called an index vector. These index vectors are sparse, high-dimensional, and ternary, which means that their dimensionality (d) is on the order of thousands, and that they consist of a small number of randomly distributed +1s and -1s, with the rest of the elements of the vectors set to 0.

- Then, context vectors are produced by scanning through the text, and each time a word occurs in a context (e.g. in a document, or within a sliding context window), that context's d-dimensional index vector is added to the context vector for the word in question. Words are thus represented by d-dimensional context vectors that are effectively the sum of the words' contexts.

The Johnson-Lindenstrauss lemma states that if points in a vector space are projected into a randomly selected subspace of sufficiently high dimensionality, the distances between the points are approximately preserved [19].

Thus, the dimensionality of a given matrix A can be reduced by multiplying it with (or projecting it through) a random matrix R:

$$A_{w\times d}R_{d\times k} = A'_{w\times k} \tag{1}$$

If the random vectors in matrix R are orthogonal, then $A = A'$; if the random vectors are nearly orthogonal, then $A \approx A'$ in terms of the similarity of their rows. A very common choice for matrix R is to use Gaussian distribution for the elements of the random vectors.

The eventual number of dimensions, and thus topics, is defined by k, and random projection does not provide an explicit way of computing it, being a parameter of the model.

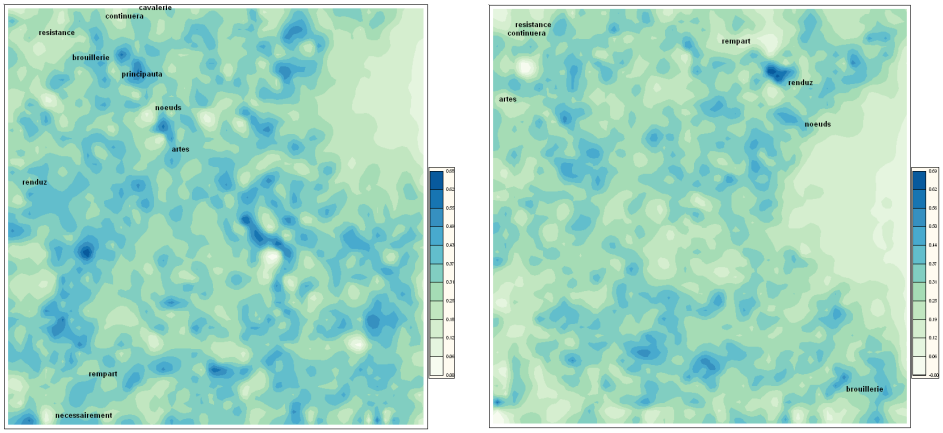
Equation 1 provides a very convenient formulation of the problem. If A is a sparse term-document matrix with w terms and d documents obtained through inverted indexing, and we generate an appropriate dense random matrix, the problem of random projection reduces to a sparse-dense matrix multiplication. The CUDA Sparse Matrix library (cuSPARSE) library distributed with CUDA provides such a call, hence a simple GPU implementation is almost trivial.

We extended the GPU version to work on distributed nodes with MR-MPI. The distribution of work is based on term space of the term-document matrix: a work node is assigned a fixed number of rows (terms) of the term-document matrix by the map call.

4.3. Self-organizing maps

The self-organizing map (SOM) training algorithm constructs a nonlinear and topology preserving mapping of the input data set $X = \{x(t)|t \in \{t_{0,\dots,t_f}\}\}$, where t_0 and t_f are the beginning and the end of the current epoch, onto a set of neurons $M = n_1, \dots, n_k$ of a neural network with associated weight vectors $W = w_1(t), ..., w_k(t)$ [23] at a given time step t. Each data point $x(t)$ is mapped to its best match neuron $\mathbf{bm}(x(t)) = n_b \in M$ such that $d(x(t), w_b(t)) \leq d(x(t), w_j(t)) \quad \forall w_j(t) \in W$, where d is the distance on the data set. The neurons are arranged on a two dimensional map: each neuron i possesses a set of two coordinates embedded in a two dimensional surface. Next the weight vector of the best match neuron and its neighbours are adjusted toward the input pattern using the following equation:

$$w_j(t+1) = w_j(t) + \alpha h_{bj}(t)[x(t) - w_j(t)],$$



(a) A self-organizing map trained on a collection of scientific correspondence at the end of the first period

(b) A self-organizing map trained on a collection of scientific correspondence at the end of the second period

where $0 < \alpha < 1$ is the learning factor, and $h_{bj}(t)$ is the neighbourhood function that decreases for neurons further away from the best match neuron in grid coordinates. A frequently used neighbourhood function is the Gaussian:

$$h_{bj} = \exp(\frac{-||r_b - r_j||}{\delta(t)}),$$

where r_b and r_j stand for the coordinates of the respective nodes. The width $\delta(t)$ decreases from iteration to iteration to narrow the area of influence. It is assumed, that the SOM gives a mapping with minimal, or at least tolerable, topological errors [22]. The training might be repeated again on the same data set to increase the fit, a training cycle is referred to as an epoch. Eventually, the neighbourhood function decreases to an extent that training might stop. The time needed to train an SOM grows linearly with the dataset size and it also grows linearly with the number of neurons in the SOM.

A finished map is an excellent and widely used tool to give a visual representation of topics and their relations in a body of text. It is even more useful to illustrate temporal changes as the topics shift over time. An example is shown in Figure 4.2, where we trained the maps on a collection of scientific correspondence on two subsequent time periods to study topic

Algorithm 1 Calculate a Euclidean distance matrix with matrix operations ($\circ$ is the Hadamard product)

1: $v_1 = (X \circ X)[1, 1 \dots 1]'$
2: $v_2 = (W \circ W)[1, 1 \dots 1]'$
3: $P_1 = [v_1 v_1 \dots v_1]$
4: $P_2 = [v_2 v_2 \dots v_2]'$
5: $P_3 = XW'$
6: $D = (P_1 + P_2 - 2P_3)$

development[1].

In the batch formulation of SOM training, the weight vectors are only updated all at once by the end of a learning period after seeing the complete set of training vectors. The new weights are calculated according to:

$$w_j(t_f) = \frac{\sum_{t'=t_0}^{t_f} h_{bj}(t')x(t')}{\sum_{t'=t_0}^{t_f} h_{bj}(t')}. \tag{2}$$

The most time-consuming part of the calculations is finding the best matching neuron. This involves calculating the pairwise distances between the elements of the data sets and the weight vectors. Euclidean distance is one of the most common distances used in SOMs. The distance between a data point and a weight vector in the batch formulation can be calculated by

$$d(w_j(t_0), x(t)) = \sqrt{\sum_{i=1}^{N} (x_i(t) - w_{ji}(t_0))^2}, \tag{3}$$

where N is the dimension of the space that embeds the data set, and $t \in \{t_0, \dots, t_f\}$. The pairwise distances can be efficiently computed on a GPU if the distances are calculated on the same space [10]. The above formula is more general, and a much higher performance can be achieved if the entire distance matrix is calculated, and the steps are decomposed into matrix level operations (see Algorithm 1 [28, 42]).

[1] An animated sequence is also available at `http://www.youtube.com/watch?v=sTR6yX0Ki9U`

The algorithm does not calculate the Euclidean distances, but the square of the distances. Taking the square root is an expensive operation on the GPU, and since we seek the minimum of the distances, we can omit calculating it. The experimental results in [28] have shown a speed up of 15x on datasets which contain more than half million data points with the above algorithm on the GPU.

An important point to note is that the norms of X do not change between subsequent epochs, these values can be computed before the main training loop starts. Step 5 is a BLAS matrix multiplication, which can be calculated on the GPU with a high-level CUBLAS call. The CUBLAS library is distributed with CUDA, and it may not be the fastest implementation at a given time, but it gives an optimized performance [3].

The other steps have to be implemented carefully to minimize global memory access and avoid bank-conflicts in the shared memory of SMPs. This is achieved by using a column-major representation of the matrices. The minimum is found by a multi-step reduction algorithm. Existing GPU-based SOM algorithms take a similar approach in finding the best matching unit, but they do not necessarily use matrix operations to derive the distance matrix [34, 46].

A distributed, MapReduce-based SOM also builds on the batch formulation described in Equation 2 [48]. The implementation builds on MR-MPI. The key idea is that before the data set is partitioned by a map call, the current set of weight vectors is broadcast to all worker nodes. The update in Equation 2 is performed locally at each node, and the reduce step sums up the updates, and eventually the master nodes sets the values of the new weight vectors. The process repeats with subsequent epochs.

We extended the MapReduce SOM algorithm by moving all calculations on local nodes to the GPU with the matrix-based Euclidean distance matrix and reduction algorithm described above. The chunk of X assigned to the node and the corresponding norms of X are kept in the GPU memory between subsequent epochs, and the weight vectors are copied to the GPU memory in each step after the node receives the broadcast of the update.

5. Discussion of experimental results

5.1. Cluster Configuration

We built a Beowulf cluster of eight nodes using Amazon Web Services (AWS). AWS ensures that cluster instances that are launched simultaneously

are physically close. The cluster instances are connected with a 10 Gigabit Ethernet network. A cluster compute GPU instance in AWS has two Intel Xeon X5570 quad-core CPUs and 23 GB of memory, and it is equipped with two NVidia Tesla M2050. One Tesla card has 448 CUDA cores and 3GByte of device memory.

The implementation of the algorithms[2], except for creating the inverted index, used OpenMPI 1.5, the June 2011 version of MR-MPI, and CUDA 4.0. The inverted index was created using Lucene 3.4 and was not timed.

5.2. Data set

The collection consists of 84,283 PhD theses crawled from the database of the German Nationaly Library. The files are in PDF format and the total size is 498GByte. The file size varies widely, averaging around 6Mbytes, but many files are over 100MBytes. The collection is multilingual, with the majority of documents being in English or German.

We created an inverted index with Lucene, applying PDF extraction. We did not use stemming or any other language specific processing step. The merged and optimized index was approximately 11GByte, with a total of 34,965,200 terms. We also ran experiments with a subset, which consisted of 1/10th of all terms.

5.3. Running time of random projection

The CPU version of random projection used version 2.4 of the SemanticVectors package [53]. This is a single-core implementation, and hence it is not entirely fair to compare it to a distributed algorithm, yet we believe it gives a good indication of the CPU performance. Since there is little data dependence, the speedup should be close to linear if the algorithm would use several cores. SemanticVectors is written in Java, and almost the entire main memory has to be reserved as heap memory to finish the projection.

The GPU implementation generated the random vectors on the CPU. While this could be done on the GPU, the operation includes many conditional branches, and the time taken is negligible, hence it matters very little whether the CPU or GPU performs this operation. This reduced the GPU implementation to a few memory copies and a sparse matrix multiplication

[2]The GPU-based SOM is available at `https://github.com/peterwittek/mr-mpi-som-gpu`

Table 1: Speedup of GPU variant of random projection over the CPU variant as the function of the number of GPUs. Note that the baseline is a single-core CPU implementation.

GPUs		2	4	8	16
All terms	With I/O				5.10
	Projection only				19.37
Subset	With I/O	2.34	3.46	4.53	5.38
	Projection only	2.02	4.05	8.10	16.45

with cuSPARSE. Since the result matrix is dense, and all matrices involved have to fit the GPU memory, all sixteen Tesla cards had to contribute to perform the projection. Each task was assigned 2,185,325 terms, save for the last one which had less. The dimension of the projection was one hundred in both the CPU and the GPU cases.

The total wall time of the MPI execution of the GPU implementation was 167.70 seconds (Table 1). In this case, initializing MPI, setting up the CUDA context, and memory transfers take up most of the time. This execution time compares to the CPU version's 856 seconds. This translates to a speedup of 5.10, which is not promising, considering that there were eight nodes and a total of sixteen GPUs involved.

If the result matrix is kept in the GPU memory for further processing and it is not saved, the execution time of the GPU version reduces to a bare 82.71s. This means a speedup of 10.34, which is certainly better.

The average GPU kernel execution time was 26.73 seconds. If we focus on the projection part of the CPU implementation, it finishes in 532 seconds, which gives a speedup of 19.91. This is the theoretical maximum speedup, which assumes that the CUDA contexts were already initialized and that the data already resided in the device memory. Looking at the profiler output, we noticed that the GPU occupancy of the sparse matrix multiplication was only 50%. We believe that the speedup could be higher if the implementation was optimized better.

The low saturation of GPU shows even better when we look at only a subset of the collection. The speedup of the projection dropped further, to barely 2x when using two GPUs.

Table 2: Speedup of GPU variant of self-organizing maps over the CPU variant as the function of the number of GPUs. Note that the baseline is the cluster of the same size, that is, two GPUs are compared to four CPU cores, four GPUs are compared to eight CPU core, etc.

GPUs		2	4	8	16
All terms	With I/O				8.69
	One epoch				9.68
Subset	With I/O	8.57	7.49	6.48	4.85
	One epoch	9.68	9.42	9.75	9.56

5.4. Running time of self-organizing maps

The baseline distributed, multicore CPU implementation was based on MR-MPI-SOM [48]. We used all sixty-four physical cores across the eight nodes. When scaling up, we noticed a nearly linear scaling, which shows that the overhead of broadcasting the SOM weight vectors is not considerable (Table 2).

Just like in the case of random projection, we had to use all sixteen GPUs to fit the problem in the distributed device memories. Even then, the tensor describing the SOM at a given time was a serious limiting factor, and we had to limit our experiments to a small 10x10 map in both the CPU and GPU cases. The bulk of the memory is taken up by the dense matrix chunk that is assigned to a GPU. Since such a tiny map does not show emergent clustering features very well, in the future we intend to work on a variant of random projection that keeps the resulting structure sparse.

The total wall time of the CPU variant was 18977.10 seconds. The wall time of the GPU implementation was 2184.4 seconds, including initializing the CUDA context, and memory transfers before and after the SOM training. This translates to a speedup of 8.69x. Considering that this is a speedup over sixty-four cores of CPUs, we believe that this is a significant result.

Turning our attention to the execution time of one epoch, a CPU core finished its chunk in 18817.1 seconds. The GPU variant took 1945.4 seconds, which means a speedup of 9.68x. Note that we only calculated the best matching units on the GPU, so there is room for further improvement.

Given the volume of data, the GPUs easily became saturated, as it can be easily seen from the speedups on the subset of the collection. Here, compared to a CPU cluster of equivalent size, the speedup was farely constant. The

Table 3: Lines of code

	MapReduce-related	CPU variant	GPU code
Random projection	28	N/A	182
SOM	35	39	296

overhead of reading and writing the data remained constant, as the master node did all the I/O and we did not use a distributed filesystem.

Looking at the GPU occupancy results, we were not far from fully loading the stream processors. Steps 1 and 2 of Algorithm 1, that is, calculating the norms, were performed with 100% occupancy. Finding the minimum is also optimal. The CUBLAS dense matrix multiplication proved to be the weak point, with an occupancy varying between 33% and 83%.

5.5. Lines of code

Lines of code measures the size of a software program by counting the number of lines in the text of the source of the program. As a metric it is often used to indicate the complexity or amount of effort that is required to develop a program. We give estimates of how long certain parts of the code are in Table 3.

The MapReduce-related code is quite minimal in either algorithm, which only proves how efficient the framework is. Self-organizing maps need some extra broadcast and communication calls, these were counted towards the MapReduce code, although not strictly related.

It is more disheartening that despite the recent developments in GPU computing, the GPU code is still 7.5x longer than the equivalent part in the CPU code (note that the CPU version of random projection uses a very different method and is implemented in a different language, and therefore it is not directly comparable to the GPU code). Considering that standard BLAS calls were also utilized, we must conclude that efficiently leveraging on the power of GPUs still takes a considerable effort.

Generally speaking, GPU programmability improves with each subsequent hardware generation. Moreover, there are several initiative that attempt to make coding simpler. OpenACC, for instance, enables the offloading of loop-level parallelism and regions in C/C++ and Fortran code to hardware accelerators. It is an initiative by hardware and compiler vendors to have a unified syntax of various directive-based approaches for accelerators. First

experiments show that execution time can be barely 40 % of a hand-written kernel [54], although at the fraction of coding effort (46 lines of OpenACC directives as opposed to 630 lines of OpenCL code).

While OpenACC is restricted to a single-node heterogeneous system, OmpSs goes beyond and promises task-based parallelism in a distributed heterogeneous system [6]. OmpSs is a variant of OpenMP which also takes a directive-based approach to parallelism. It handles heterogeneous hardware that includes GPUs and CPUs, and takes care of the memory copies based on explicitly expressed data dependencies. Asynchronous communication can also be defined as a task. It still requires writing kernels, but theoretically it should be able to simplify communication.

6. Conclusions

Text mining applications are both data- and compute-intensive, but the two types of load can be separated in many scenarios. Data-intensive tasks, or tasks that involve a large number of sequential algorithms that are hard to parallelize can be run on CPU instances in cluster, whereas compute-intensive tasks can be executed on GPUs. We give an example how text mining workflow, consisting of inverted indexing, random projection, and self-organizing maps, can be accelerated using a small, GPU-equipped cluster. We use two MapReduce frameworks for the deployment of the steps. This is particularly important because MapReduce jobs are fairly easy to develop, a wide range of machine learning algorithms has already been adapted to this paradigm, hence developers of text mining applications will find it convenient to build novel, computationally demanding services for the end-users. In particular, our key contributions are as follows:

- Entirely MapReduce-based, flexible and scalable text mining workflow;
- A GPU-based, distributed random indexer;
- A GPU version of a distributed SOM algorithm.

Our ongoing work revolves around communication across the distributed GPUs. MapReduce as a framework is not very efficient in overlapping communication with computation, and we are looking at methods how to overcome this problem. One vendor-specific solution is by Nvidia, the so-called GPU Direct technology that enables faster communication within a node between multiple GPUs, and also across different nodes. The latest advances

in the GPU architecture also enable kernel launches from a kernel, making the GPUs autonomous computing devices, as opposed to accelerators. We are investigating what it means with regard to subsequent MapReduce jobs.

7. Acknowledgment

This work was sponsored by Sustaining Heritage Access through Multivalent ArchiviNg (SHAMAN), a large-scale integrating project co-funded by the European Union (Grant Agreement No. ICT-216736). This work was also supported by Amazon Web Services.

References

[1] , 2011. NVida Compute Unified Device Architecture C Best Practices Guide 4.0.

[2] Bai, B., Weston, J., Grangier, D., Collobert, R., Sadamasa, K., Qi, Y., Chapelle, O., Weinberger, K., 2009. Supervised semantic indexing, in: Proceeding of CIKM-09, 18th ACM Conference on Information and Knowledge Management, Hong Kong. pp. 187–196.

[3] Barrachina, S., Castillo, M., Igual, F., Mayo, R., Quintana-Orti, E., 2008. Evaluation and tuning of the level 3 CUBLAS for graphics processors, in: Proceedings of IPDPS-08, 22nd International Symposium on Parallel and Distributed Processing, Miami, FL, USA. pp. 1–8.

[4] Blei, D., Ng, A., Jordan, M., 2003. Latent Dirichlet allocation. The Journal of Machine Learning Research 3, 993–1022.

[5] Brodtkorb, A., Dyken, C., Hagen, T., Hjelmervik, J., Storaasli, O., 2010. State-of-the-art in heterogeneous computing. Scientific Programming 18, 1–33.

[6] Bueno, J., Martinell, L., Duran, A., Farreras, M., Martorell, X., Badia, R., Ayguade, E., Labarta, J., 2011. Productive cluster programming with OmpSs, in: Proceedings of Europar-11, Bordeaux, France. pp. 555–566.

[7] Byna, S., Meng, J., Raghunathan, A., Chakradhar, S., Cadambi, S., 2010. Best-effort semantic document search on GPUs, in: Proceedings of GPGPU-10, 3rd Workshop on General-Purpose Computation on Graphics Processing Units, New York, NY, USA. pp. 86–93.

[8] Cafarella, M., Cutting, D., 2004. Building Nutch: Open source search. Queue 2, 54–61.

[9] Cavanagh, J., Potok, T., Cui, X., 2009. Parallel latent semantic analysis using a graphics processing unit, in: Proceedings of GECCO-09, 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers, Montreal, QC, Canada. pp. 2505–2510.

[10] Chang, D., Jones, N., Li, D., Ouyang, M., Ragade, R., 2008. Compute pairwise Euclidean distances of data points with GPUs, in: Proceedings of CBB-08, International Symposium on Computational Biology and Bioinformatics, ACTA Press, Orlando, FL, USA. pp. 278–283.

[11] Charles, J., Potok, T., Patton, R., Cui, X., 2007. Flocking-based document clustering on the graphics processing unit, in: Proceedings of NICSO-07, 2nd International Workshop on Nature Inspired Cooperative Strategies for Optimization, Acireale, Italy. pp. 27–37.

[12] Daróczy, B., Pethes, R., Benczúr, A., 2011. SZTAKI @ ImageCLEF 2011, in: Proceedings of CLEF-11, Conference on Multilingual and Multimodal Information Access Evaluation, Amsterdam, The Netherlands.

[13] Dean, J., Ghemawat, S., 2004. MapReduce: Simplified data processing on large clusters, in: Proceedings of OSDI-04, 6th International Symposium on Operating Systems Design & Implementation, San Francisco, CA, USA.

[14] Deerwester, S., Dumais, S., Furnas, G., Landauer, T., Harshman, R., 1990. Indexing by latent semantic analysis. Journal of the American Society for Information Science 41, 391–407.

[15] Ding, S., He, J., Yan, H., Suel, T., 2009. Using graphics processors for high performance IR query processing, in: Proceedings of WWW-09, 18th International Conference on World Wide Web, Spain, Madrid. pp. 421–430.

[16] Gospodnetic, O., Hatcher, E., et al., 2005. Lucene in Action. Manning.

[17] He, B., Fang, W., Luo, Q., Govindaraju, N., Wang, T., 2008. Mars: A MapReduce framework on graphics processors, in: Proceedings of PACT-08, 17th International Conference on Parallel Architectures and Compilation Techniques, Toronto, ON, Canada. pp. 260–269.

[18] Jiménez, V., Vilanova, L., Gelado, I., Gil, M., Fursin, G., Navarro, N., 2009. Predictive runtime code scheduling for heterogeneous architectures. High Performance Embedded Architectures and Compilers , 19–33.

[19] Johnson, W., Lindenstrauss, J., 1984. Extensions of Lipschitz mappings into a Hilbert space. Contemporary Mathematics 26, 189–206.

[20] Kanerva, P., Kristofersson, J., Holst, A., 2000. Random indexing of text samples for latent semantic analysis, in: Proceedings of CogSci-00, 22nd Annual Conference of the Cognitive Science Society, Philadelphia, PA, USA.

[21] Kirk, D., Hwu, W., 2009. Programming massively parallel processors: A hands-on approach .

[22] Kohonen, T., 2001. Self-Organizing Maps. Springer.

[23] Kohonen, T., Kaski, S., Lagus, K., Salojärvi, J., Honkela, J., Paatero, V., Saarela, A., 2000. Self organization of a massive text document collection. IEEE Transactions on Neural Networks 11, 574–585.

[24] Koop, M., Sur, S., Gao, Q., Panda, D., 2006. High performance MPI design using unreliable datagram for ultra-scale InfiniBand clusters, in: Proceedings of ISC-06, 21st Annual International Conference on Supercomputing, Dresden, Germany. pp. 180–189.

[25] Kuhn, B., Petersen, P., O'Toole, E., 2000. OpenMP versus threading in C/C++. Concurrency: Practice and Experience 12, 1165–1176.

[26] Lahabar, S., Narayanan, P., 2009. Singular value decomposition on GPU using CUDA, in: Proceedings of IPDPS-09, 23rd International Symposium on Parallel and Distributed Processing, Rome, Italy.

[27] Lee, S., Min, S., Eigenmann, R., 2009. OpenMP to GPGPU: a compiler framework for automatic translation and optimization, in: Proceedings of PPOPP-09, 14th Symposium on Principles and Practice of Parallel Programming, pp. 101–110.

[28] Li, Q., Kecman, V., Salman, R., 2010. A chunking method for Euclidean distance matrix calculation on large dataset using multi-GPU, in: Proceedings of ICMLA-10, 9th International Conference on Machine Learning and Applications, Washington, DC, USA. pp. 208–213.

[29] Lin, J., 2008. Scalable language processing algorithms for the masses: A case study in computing word co-occurrence matrices with MapReduce, in: Proceedings of EMNLP-08, 13th Conference on Empirical Methods in Natural Language Processing, Honolulu, HI, USA. pp. 419–428.

[30] Luk, C., Hong, S., Kim, H., 2009. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping, in: MICRO-42, 42nd Annual IEEE/ACM International Symposium on Microarchitecture, New York, NY, USA. pp. 45–55.

[31] Masada, T., Hamada, T., Shibata, Y., Oguri, K., 2009. Accelerating collapsed variational Bayesian inference for latent Dirichlet allocation with NVidia CUDA compatible devices, in: Proceedings of IEA/AIE-09, 22nd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, Tainan, Taiwan. pp. 491–500.

[32] Michael, M., Moreira, J., Shiloach, D., Wisniewski, R., 2007. Scale-up x scale-out: A case study using Nutch/Lucene, in: Proceedings of IPDPS-07, 21st International Parallel and Distributed Processing Symposium, Long Beach, CA, USA. pp. 1–8.

[33] Moreira, J., Michael, M., Da Silva, D., Shiloach, D., Dube, P., Zhang, L., 2007. Scalability of the Nutch search engine, in: Proceedings of ICS-07, 21st Annual International Conference on Supercomputing, Seattle, WA, USA. pp. 3–12.

[34] Oh, K., Jung, K., 2004. GPU implementation of neural networks. Pattern Recognition 37, 1311–1314.

[35] Owen, S., Anil, R., Dunning, T., Friedman, E., 2010. Mahout in Action. Manning Publications Co.

[36] Owens, J., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., Purcell, T., 2007. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26, 80–113.

[37] Phillips, J., Stone, J., Schulten, K., 2008. Adapting a message-driven parallel application to GPU-accelerated clusters, in: Proceedings of SC-08, 21st Conference on Supercomputing, Austin, TX, USA. pp. 1–9.

[38] Plimpton, S., Devine, K., 2011. MapReduce in MPI for large-scale graph algorithms. Parallel Computing .

[39] Puzovic, N., 2012. Mont-Blanc: Towards energy-efficient HPC systems, in: Proceedings of CF-12, 9th International Conference on Computing Frontiers, Cagliari, Italy. pp. 307–308.

[40] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C., 2007. Evaluating MapReduce for multi-core and multiprocessor systems, in: Proceedings of HPCA-07, 13th International Symposium on High Performance Computer Architecture, Phoenix, AZ, USA. pp. 13–24.

[41] Sahlgren, M., 2005. An introduction to random indexing, in: Proceedings of TKE-05, Methods and Applications of Semantic Indexing Workshop at the 7th International Conference on Terminology and Knowledge Engineering, Copenhagen, Denmark.

[42] van de Sande, K., Gevers, T., Snoek, C., 2011. Empowering visual categorization with the GPU. IEEE Transactions on Multimedia 13, 60–70.

[43] Sanderson, R., Watry, P., 2007. Integrating data and text mining processes for digital library applications, in: Proceedings of JCDL-07, 7th ACM/IEEE-CS Joint Conference on Digital Libraries, Vancouver, Canada. pp. 73–79.

[44] Shirahata, K., Sato, H., Matsuoka, S., 2010. Hybrid map task scheduling on GPU-based heterogeneous clusters, in: Proceedings of CloudCom-10, The 2nd International Conference on Cloud Computing, Indianapolis, IN, USA.

[45] Stone, J., Hardy, D., Ufimtsev, I., Schulten, K., 2010. GPU-accelerated molecular modeling coming of age. Journal of Molecular Graphics and Modelling 29, 116–125.

[46] Strong, G., Gong, M., 2008. Browsing a large collection of community photos based on similarity on GPU. Advances in Visual Computing , 390–399.

[47] Stuart, J., Owens, J., 2011. Multi-GPU MapReduce on GPU clusters, in: Proceedings of IPDPS-11, 25th International Parallel and Distributed Computing Symposium, Anchorage, AK, USA.

[48] Sul, S., Tovchigrechko, A., 2011. Parallelizing BLAST and SOM algorithms with MapReduce-MPI library, in: Proceedings of IPDPS-11, 25th International Parallel and Distributed Computing Symposium, Anchorage, AK, USA. pp. 476–483.

[49] Talbot, J., Yoo, R., Kozyrakis, C., 2011. Phoenix++: modular MapReduce for shared-memory systems, in: Proceedings of MapReduce-11, 2nd International Workshop on MapReduce and Its Applications, San Jose, CA, USA. pp. 9–16.

[50] Thibodeau, P., 2011. Cray to build 20-petaflop system. Computerworld 45, 2–2.

[51] Ufimtsev, I., Martínez, T., 2008. Graphical processing units for quantum chemistry. Computing in Science & Engineering 10, 26–34.

[52] Wei, Z., JaJa, J., 2011. A fast algorithm for constructing inverted files on heterogeneous platforms, in: Proceedings of IPDPS-11, 25th International Parallel and Distributed Computing Symposium, Anchorage, AK, USA.

[53] Widdows, D., Ferraro, K., 2008. Semantic vectors: a scalable open source package and online technology management application, in: LREC-08, 6th International Conference on Language Resources and Evaluation, Marrakech, Morocco.

[54] Wienke, S., Springer, P., Terboven, C., an Mey, D., 2012. OpenACC–first experiences with real-world applications, in: Proceedings of EuroPar-12, Rhodes Island, Greece.

[55] Wong, S., Ziarko, W., Wong, P., 1985. Generalized vector space model in information retrieval, in: Proceedings of SIGIR-85, 8th International Conference on Research and Development in Information Retrieval, Montréal, Québec, Canada. pp. 18–25.

[56] Wu, R., Zhang, B., Hsu, M., 2009. Clustering billions of data points using gpus, in: Proceedings of UHPC-11, 2nd Workshop on Unconventional High Performance Computing, Ischia, Italy. pp. 1–6.

[57] Yan, F., Xu, N., Qi, Y., 2009. Parallel inference for latent Dirichlet allocation on graphics processing units. Advances in Neural Information Processing Systems 22.

[58] Yoo, R., Romano, A., Kozyrakis, C., 2009. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system, in: Proceedings of IISWC-09, 4th International Symposium on Workload Characterization, Austin, TX, USA.

[59] Zhang, Y., Mueller, F., Cui, X., Potok, T., 2010. Large-scale multi-dimensional document clustering on GPU clusters, in: Proceedings of IDPDS-10, 24th International Parallel and Distributed Computing Symposium, Atlanta, GA, USA.