A Framework for Direct and Transparent Data Exchange of Filter-stream Applications in Multi-GPUs Architectures

Gabriel Ramos[1], Guilherme Andrade[2], Rafael Sachetto[1], Daniel Madeira[1]
, Renan Carvalho[1], Renato Ferreira[2], Fernando Mourão[1], and Leonardo Rocha[1]

[1] Universidade Federal de São João del Rei, Brazil
{gramos,sachetto,rcarvalho,fhmourao,lcrocha}@ufsj.edu.br
[2] Universidade Federal de Minas Gerais, Brasil
{gnandrade,renato}@dcc.ufmg,br

Abstract
The massive data generation has been pushing for significant advances in computing architectures, reflecting in heterogeneous architectures, composed by different types of processing units. The filter-stream paradigm is typically used to exploit the parallel processing power of these new architectures and the efficiency of applications is achieved by exploring in a coordinated way a set of interconnected computers (cluster) using filters and communication between them. In this work we propose, implement and test a generic abstraction for direct and transparent data exchange of filter-stream applications in heterogeneous cluster with multi-GPU (Graphics Processing Units) architectures. This abstraction allows all the low-level implementation details related to GPU communication and the control related to the location of the filters to be performed transparently to the programmers. This work is consolidated in a framework and our evaluation results show that it provides an abstraction layer without compromising the overall application performance.

Keywords: Multiple GPUs, Filter Stream Application, Heterogeneous Architectures

1 Introduction

The development of new technologies has been characterized by the massive data generation. The continuous data growth, associated with more sophisticated processing in different areas of knowledge, has been pushing for significant advances in computing architectures, reflecting in more efficient storage systems, as well as the use of different types of processing units, in the so-called heterogeneous architectures. One example of these new architectures are computers with multiple processors (multicore architecture) and graphics cards (using CUDA architecture [12] for example). In this context, in addition to the traditional multicore parallel programming interfaces [8, 10], we have recently seen the popularization of new libraries and environments [12, 15, 18] that allow the development of applications in alternative processing

units such as Graphics Processing Units (GPUs). These advances have been essential to enable different algorithms to exploit the parallel processing power of these new architectures, making them feasible for large volumes of data.

Despite these advances, there are scenarios where a single efficient parallel implementation is not enough. Examples of such scenarios are applications composed of different processing stages, such as image segmentation and knowledge discovery in databases [3], where each stage receives a given input and produces an output that will be used as input to another stage. These applications are typically modeled using the filter-stream paradigm [1], whose processing steps are represented by filters, which are instantiated on one or more machines, and the data passing between these steps is represented by data streams. The efficiency of applications implemented using the filter-stream paradigm is achieved, mainly, by the parallel distribution of these filters, exploring in a coordinated way a set of interconnected computers (cluster computing). Consequently, the performance of these applications is directly dependent on the efficient communication between the filters.

Traditionally, in the filter-stream execution environments found in the literature the communication is made mainly using the Message Passing Interface (MPI) [10]. The MPI standard has been shown to be efficient in most filter-stream scenarios, and can be applied to the communication between instantiated filters on the same machine or on different machines. However, when the cluster used to run an application has heterogeneous machines, in which a particular filter can be processed in a GPU for example, the communication via MPI between filters may require some extra data movement in order to be completed. In this case, the data to be transmitted must be first transferred from the GPU's global memory to the computer's primary memory, and then sent via MPI to the destination filter. Similar movement should occur in the destination filter if it is also processed in a GPU. These extra data movements between processing unit memories can compromise the application performance. To the best of our knowledge, no current filter-stream parallel programming environment allows direct communication between GPUs.

In order to avoid these excessive movements, the GPUs responsible for processing the filters must be able to communicate directly, without the need to copy the data to the CPU memory. There are strategies in the literature that make this direct communication between GPUs possible, for example: (i) GPUDirect RDMA [14] for transferring data between GPUs located at the same machine; (ii) Cuda Aware MPI (OpenMPI + GPUDirect RDMA) [19] mainly focusing on communication between GPUs on different machines. Moreover, in the filter-stream scenario, this direct communication is a greater challenge, since filters can be instantiated in the same machine or in different machines, making the programmer responsible for controlling where each filter is instantiated and deciding which strategy to be used to perform the communication between filters running in different GPUs. In addition to being a manual control for the programmer, it is an approach that needs to be adjusted and re-implemented at each new filter locale setting.

Thus, in this work we propose a generic abstraction for direct data exchange between GPUs in filter-stream scenarios. Our proposal makes transparent to the programmer the low-level implementation details related to GPU communication and furthermore removes from the programmer the responsibility to control the location of the filters and to choose which communication strategy to use. We consolidated our proposal into a filter-stream framework, which also provides an API to assist in the implementation of filters, as well as establishing a standard for representing data flow between filters. In our evaluation, first we compare different communication strategies between GPUS, intra-node and inter-node, in order to justify the choices made for our framework. Then, we evaluate the efficiency of our proposal implementing a real application using our framework. Comparing this implementation with one using directly the existing technologies to change data between GPUs, we show that the execution time of real

application is the same for both implementation. This fact demonstrates that our proposal, besides to provide an abstraction layer to facilitate the programming process, as well automatically controls the instantiation of filters, does not compromise the application performance, which correspond to our main contribution.

2 Related Work

The filter-stream paradigm [1] is an important tool to support the efficient development of a sort of applications, such as image segmentation and knowledge discovery in databases [3], in computer distributed systems. This paradigm is able to work, in a coordinated way, with a set of interconnected computers (cluster computing), exploring the individual performance of each computer, as well the parallel collaboration between them. In order to model an application in filter-stream paradigm, the stages are represented by filters which can be instantiated in one or more computers. In turn, the stream consists of the communication between filters, sending or receiving data related to the different stages of the application. In this paper, we are focused on a solution for this niche of application.

We can find in literature various proposals of environments that provide efficient tools to deal with different challenges related to the implementation of this category of application, such as instantiation management of filters, communications between filters, etc [9, 6, 2]. In citede2011watershed is presented the Watershed, a filter-stream environment that provides a C++ API for the implementation of each filter. Moreover, the stream of data between filters can be described using XML (eXtensible Markup Language) files. Furthermore, in these XML files is possible to configure which computers will be responsible to execute each filter. Therefore, Watershed consists of a complete execution environment on which the programmers should only load the configuration files and the implemented filters, all the execution is performed in a transparent way. Similar proposal is present in [6], the Datacuter, with practically the same functionalities. Finally, in [2], is proposed a system that allows to save the state of applications in distributed systems, exploring dynamic configurations in a cluster. In this paper, we are not proposing a complete environment as the ones described in this paragraph. However, our generic abstraction for direct data exchange between GPUs on filter-stream scenarios can be adapted by all of these proposals..

Recently, the parallelization of applications using GPUs have been presented itself as a promissory research topic [13, 11, 5, 17]. Despite that, in scenarios of filter-stream application, to the best of our knowledge, we do not find filter-stream execution frameworks that allow the use of GPUs. This happens due to limitations related to the existing technologies for sharing efficiently data between different GPUs: the need for low-level implementation details related to GPU communication and the complexity to control the instantiation of filters. There are, basically, two different approaches to deal with this question: (1) intra-node strategies, which are exclusively focused in share data between GPUs presented in the same computer [7, 16, 20]; and (2) inter-node strategies, focused in solutions for GPUs presented in different computers [14, 4, 19].

Regarding the intra-node strategies, a common technology used is the *Unified Virtual Addressing* (UVA), that allows different processing units (i.e. CPU and GPU) access the same memory address space, reducing the total of memory copies. In [16] is presented an optimization on which the data that are in different address spaces are encapsulated in a single object in order to improve the performance of applications with intensive memory access. Moreover, in [20], the authors present a new system for re-allocate memory in clusters based on the single address space concept. Recently, the implementation of this concept specifically for data changes between GPUs was improved by GPUDirect RDMA [14], a new technology that allow

the exchange of data directly by PCI Express Bus. In [7] the authors present a detailed study of the performance of this technology. Using real applications, such as image processing applications and mathematical simulations, different aspects are evaluated on which the applications achieved good results in terms of efficiency, associated with simple implementations.

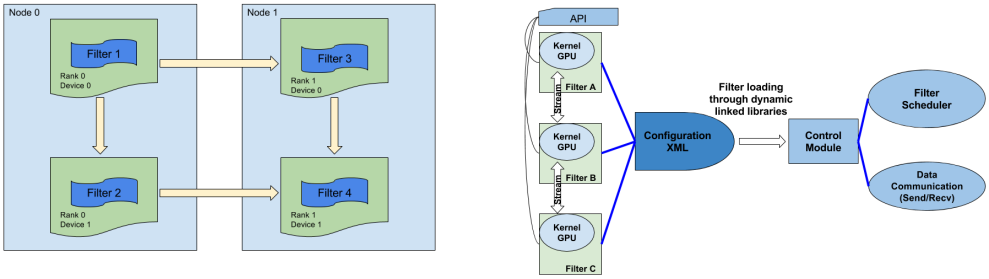
The approaches related to inter-node [4, 19] are based on the combination of GPUDirect RDMA technology and implementations of MPI standard [10], such as OpenMPI (i.e. CUDA-aware MPI). In these approaches, instead to use the PCI Express Bus in order to exchange data between GPUs, the network interface is used and the communication process between different computers is performed by OpenMPI. In [4] the authors prepared a benchmark that was used in order to evaluate different strategies for transferring data between GPUs presented in different computers. The evaluation was performed in a Infinitband cluster and the results showed that the combination of GPUDirect RDMA technology and OpenMPI was up to 50% more efficient that the other ones strategies.

In this paper we propose and evaluate a generic abstraction for direct data exchange between GPUs in filter-stream scenarios that aims to control where each filter is instantiated, deciding which strategy to be used to perform the communication.

3 Framework

In this section we detail our proposal to a generic abstraction for direct data exchange between GPUs in filter-stream scenarios which is consolidated in a framework. We first present a general overview of our framework architecture, followed by details related to the API proposed to assist filters implementation and configuration. Finally, we describe the framework engine.

3.1 Overview



(a) Example of Filter-Stream Application (b) Framework Architecture

Figure 1: Overview of Framework

In Figure 1 (a) we present an application that will be used as a base example to discuss in this section. This application contains four filters. The first filter loads the data and performs the initial processing. Next, the data is streamed to filters 2 and 3, which perform independent processing on the same input data set. At this point, these filters can be executed in parallel, since their data processing are independent. In addition, the data stream between filters 1 and 2 is local, whereas the data stream between filters 2 and 3 depends on transfer in the network. When these filters finish processing their data, it is sent to a last filter, which performs the final operations on the received data. This last filter depends on more than one of previous filters (2 and 3), so it is necessary to wait until all the previous filters to send their data. Also notice that each filter can execute pre or post processing operations, while it waits data to arrive or after their data is sent to the next.

The framework is separated into two main parts. First, we have an API, which provides a library defining functions specific for communication between the filters, as well as a XML configuration file to be used by the programmer to define the execution environment (i.e. the filters, the directory where the filters are located, the available machines IP addresses, and the number of GPUs available in each of these machines.). This entire configuration is used by the framework engine, which is responsible for compiling, allocating and running the application filters. After the definition of the filters code and the environment configuration, the engine compiles each filter as a dynamic library, and then runs the application in the specified environment. The engine is then, responsible to take care of the scheduling and communication tasks. In Figure 1 (b) we illustrate the architecture of our framework. Each filter is an independent function, containing one or more GPU kernels, and the streams are defined by the data that flows between each filter.

3.2 Framework API

3.2.1 Defining the Filters

In order to use the framework, the programmer needs to implement each filter in C++, saving each of them in separate source files. The data processing is independent: each filter receives its data, performs its pre-processing (if necessary) and goes on to process the data. At the end, it sends the data to the next filter. The tasks of receiving and sending the data are performed through the functions provided by the framework API. Listing 1 shows the API functions signatures for sending (*sendMessage*) and receiving (*recvMessage*) messages. The signature of the two functions is similar, receiving as parameters the name of the source filter, the name of the destination filter, the type of data to be sent and the data to be transmitted (buffer) and its size. The need to work in pairs, where the message sent by one filter must be received by another. Based on this information, as well as the environment configuration information, the engine recognizes where each filter and data are located and makes the transfers when needed.

```
1 //Filter sending data
sendMessage(char* src, char* dst, MPI_DATATYPE, void* buffer, size_t size);
3 //Filter receiving data
receiveMessage(char* src, char* dst, MPI_DATATYPE, void* buffer, size_t size);
```

Listing 1: Funtion calls for sending and receiving messages with our API.

Listing 2 provides an example of a filter implemented using the API. The *myFilter* function is responsible for receiving the input data. The function then invokes it's GPU kernel for data processing. At the end, a message is sent to the next filter. This message contains the data processed by the current filter, which will be the input of the next one. The communication step is optional as it depends on the existence of data preprocessed by a previous filter or the need to send the data to a next filter.

```
#include "comm/comm.h"
2
__global__ void kernel(){
4   // Kernel code to be executed on GPU
}
6
extern "C" void myFilter(){
8   //Receive the input data from the previous filter (if exists)
   recvMessage();
10  //Process the data
   kernel<<<>>>();
12  //Send the processed data to the next filter (if exists)
   sendMessage();
14 }
```

Listing 2: Single filter implemented using the framework API. Note that the send and receives are optional, since they depend on existing filters before or after.

3.2.2 Comunication Process

The main contribution of the proposed framework is the independence of spatial data location. The functions provided by the API automatically recognize whether the data is on the GPU or CPU, as also whether they are on the same node or not, and then perform the data transferring. In addition, they act as barriers between the filters executions whose inputs are outputs from previous filters. In the base example, while the data transferring between filters 1 and 2 can occur using an intra-node technology (i.e. GPUDirect RDMA), the communication between filters 1 and 3 must be performed using the network (i.e. CUDA-aware MPI).

For the communication between GPUs in the same node, there are, basically, two strategies. The first one it is to use the engine provided by the CUDA API, making a copy of the data from the first GPU to the CPU main memory and then copying that data back to the memory address in the second GPU, which we call **Standard Communication through CPU Memory**. The second one is based on **GPUDirect RDMA**, technology, introduced in CUDA 5.0, that allows direct access between the GPU and a third device through a PCI-Express bus, which could be another GPU, network interface and storage devices. However, the use of this technology is limited to the present hardware, i.e. whether the devices are not connected through a PCI-Express bus, this feature will not be enabled.

In the case of GPUs in different nodes, the communication can be performed by two strategies. In the first one, which we call **MPI Standard communication**, it makes a copy of the data from the first GPU to the CPU main memory and then sending that data over MPI to the second machine, where the data will be copied back to the memory address in the second GPU. The second one is the **CUDA-aware MPI** technology, on which the MPI can recognize the GPUs and can perform the copy between each GPU efficiently through GPUDirect RDMA. With this, it is possible to enable data copying between GPUs on different machines without going through the machines main memory controller. The data leaves a GPU direct to the network interface, is sent to the target machine and then copied directly to the target GPU. This strategy can also be used for data transfer in the same node, since the OpenMPI (MPI implementation used in this work) can also use the data bus instead to used network interface.

As we will present in section 4, we performed a series of tests to evaluate the different communication strategies described above, whether intra-note (Standard Communication through CPU Memory, GPUDirect RDMA and CUDA-aware MPI) or inter-node (MPI Standard communication and CUDA-aware MPI). After evaluating the results, we opted to simplify the process of data communication between GPUs of our framework, adopting always a strategy that uses CUDA-aware MPI, even in transfers in the same GPU.

3.2.3 Environment Configuration

Finally, the environment configuration to be used by the application must also be passed to the engine. For this, we use a XML file containing the information required by the engine. The Listing 3 shows an example environment setting. The configuration is simple and should contain information about each filter, the installation path of the framework and the machines to be used.

The filter configuration is inside the <modules> field. The <file> sub-field defines the file that contains the source code for the filter. The <func> sub-field defines the name of the function that implement the filter itself. The <include> and <libraries> sub-fields indicate paths and libraries to be included for the correct filter compilation as a dynamic library. The field <home> only indicates the path where the framework libraries are located. The <machine> field provides the required information about the execution environment. The <ipmachine> sub-field contains the IP address of each machine to be used, as well as the number of GPUs

available on that machine. Based on this XML configuration file, the engine performs the necessary tasks for correct application execution.

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <modules>
    <file name="myFilter.c"/>
    <func funcName="myFilter"/>
    <includes>
      <inc>/usr/include</inc>
    </includes>
    <libraries>
      <libPath>/usr/lib</libPath>
      <lib>m</lib>
    </libraries>
  </modules>

  <home>
    <directory dir="/opt/framework/"/>
  </home>

  <machine>
    <ipmachine ip="127.0.0.1" ngpus="4"/>
  </machine>
</config>
```

Listing 3: Configuration file

3.3 Framework Engine

The engine is responsible for instantiating the filters on each machine present in the configuration file, in addition to implementing the communication used in the application. For this, it receives as input the source codes of the filters and the XML configuration file. The first step is to compile the filters into dynamic libraries, allowing each filter to be loaded dynamically by the engine.

The next step is to configure the execution process, which is performed using the parameters of the configuration file. As previously mentioned, for both communication (intra-node and inter-node), we adopted the CUDA-aware MPI. Therefore, the number of filters defined in the file is used to create MPI processes with different ranks. At this point, each filter, then receives a single rank, which will be used for communication management. Next, the available machines and number of GPUs in each one are identified. With this information, each process is associated with a single GPU, using the *setCudaDevice* function present in the CUDA library.

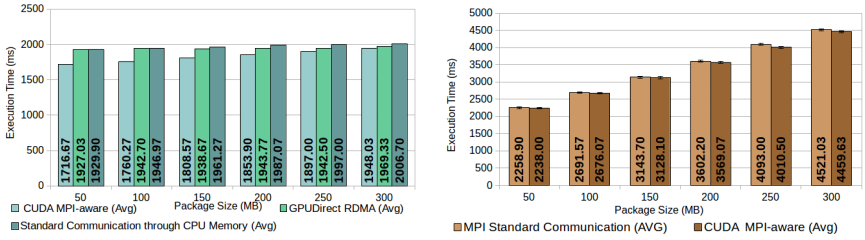
At this stage, the application starts. The engine controls the communication between each process. As each filter has a name specified in the configuration file, in addition to a single rank of its MPI process, the translation of API functions into MPI calls is done directly. Each filter runs in a MPI process, with its execution flow modeled through the barriers between data copies, defined by the framework API calls. Figure 1 (a) shows the mapping of processes ranks and their respective GPU. Filter 1, for example, is running on computer 0, has *rank* 0, and the ID of its GPU also 0. All filters run at the same time at the start of the application and wait until data is available.

4 Experimental Evaluation and Discussions

In this section we show and discuss the computational experiments that lead the decision on what communication strategies to use in our framework. Moreover, using a real application related to mathematical simulations, we evaluate the impact of the abstraction layer in the application performance. All the experiments presented in this section were performed using two GNU/Linux 4.1.13 nodes: (1) a machine with a Intel(R) Core(TM) i7-6700 3,40GHz processor, 32GB of memory and two GPUs NVIDIA GeForce GTX 1070; (2) a machine with a Intel(R) Core(TM) i5-3570 3,40GHz, 16GB of memory and a GPU NVIDIA GeForce GT 640.

4.1 Evaluation of Communication strategies

In this set of experiments we analyze the impact of communication strategies decision in the performance of our framework. More specifically, for intra-node communication we evaluate the **Standard Communication through CPU Memory**, **GPUDirect RDMA** technology and **CUDA-aware MPI technology**. For the inter-node communication we evaluated **MPI Standard Communication** and **CUDA-aware MPI technology**. We perform experiments analyzing the overhead caused by the data transfers between GPUs where the filters are running. We sent data ranging from 50MB to 300MB between the devices using the above strategies. We performed 30 runs for each of the proposed tests, in order to extract the mean and standard deviation for a better analysis of the results. The results of these experiments is presented in Figure 2.



(a) Intra-node Strategies (b) Inter-node Strategies

Figure 2: **Evaluation of Communication Strategies.** CUDA-aware MPI strategy presents better results in both scenarios.

Focusing first in intra-node tests, presented in Figure 2 (a). The CUDA-aware MPI strategy proved to be more efficient for the communication between intra-node GPUs, mainly for smaller data size, presenting a reduction up to 10.9% in the execution time compared to GPUDirect RDMA strategy, which was the second best. At first, this result could be considered surprising, since the CUDA-aware MPI is based on GPUDirect RDMA, as explained in Section 3.2.2. However, reading the extensive documentation related to OpenMPI (https://www.open-mpi.org/), it is mentioned that the OpenMPI library makes use of CUDA IPC support where possible to move the GPU data quickly between GPUs that are on the same node and same PCI root complex, using some data packaging optimizations. As we increase the size of data to be transferred, the time spent to communicate using CUDA-Aware MPI grows faster than the other strategies, since the total MPI function calls also increase. However, it is important to point the despite that, the performance of CUDA-aware MPI still better than the others strategies.

Regarding the inter-node transfers between GPUs, the results are presented in Figure 2 (b), presenting results similar to those of the intra-node strategies, but with the addition of the network communication overhead. As expected, the implementation using CUDA-aware MPI shows better execution time. The difference between these strategies was lower, presenting a reduction of only 2% compared with the strategy using MPI Standard Communication. These results justify our choice to always use the CUDA-aware MPI as our communication strategy regardless the GPU location (intra or inter-node).

4.2 Evaluation of Impact of Application Performance

In this section our goal is to evaluate the impact of the abstraction layer of our framework in the application performance. In order to do this, we consider a real application related to linear algebra that consists of filter that generate a tridiagonal sparse matrix A and a vector b and

send that to another filter that solves, using the conjugate gradient method, the linear system represented by the matrix ($Ax = b$). The solver filter, then sends the result (x) to another filter that solves the system again using the received result as the left-hand side (b) of the new system. In Figure 4 we illustrate the description of application.

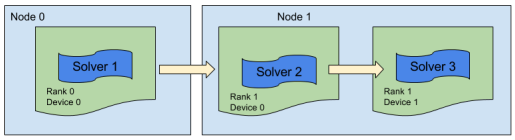


Figure 3: Filter Configuration of Real Linear Algebra Application

In this evaluation we consider two implementation of the above application: (1) one completely implemented using our framework (**Framework Implementation**), without any concern about the filter instantiation and filter communication; and (2) a **Manual Implementation**, using the functions related to CUDA-aware MPI, allocating the filters e performing the communication manually. Our goal is not to demonstrate that the performance of our framework is better than manual implementation, since its performance is limited by the adopted technologies. The results related to this evaluation are presented in Figure 4. As we can observe, the overhead related to the abstraction layer is very low, the performance of both implementations is practically equivalent. These results demonstrate that, despite the overhead related to the abstraction layer created to facilitate the programming process, as well automatically controls the instantiation of filters, it does not compromise the overall application performance.

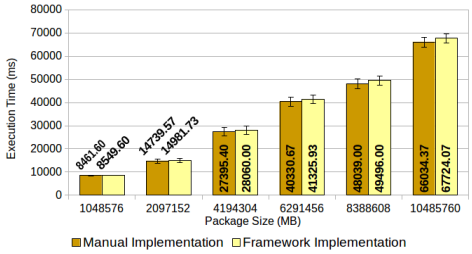


Figure 4: **Impact of Abstraction Layer in Application Performance.** As we can see, our framework does not compromise the overall application performance.

5 Conclusions and Future Work

In this work we proposed, implemented and tested a generic abstraction for direct and transparent data exchange between GPUs in filter-stream scenarios on a heterogeneous cluster, composed by nodes with multiple GPUs. This abstraction allows all the low-level implementation details related to GPU communication and the control related to the location of the filters to be performed transparently to the programmers, reducing their responsibility and allowing them to focus in details off the problem that is being solved. After evaluating several possible communication strategies we show that our proposal, besides to provide an abstraction layer to facilitate the programming process, as well automatically controls the instantiation of filters, does not compromise the overall application performance. As future works we intend to merge our framework in a complete filter-stream environment, such as Watershed [9]. We also intend to improve our framework by allowing it to verify if the available GPUs support the use of GPU Direct RDMA, and if not, chose the best available option for that hardware.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. *SIGPLAN Not.*, 33(11):81–91, Oct. 1998.

[2] A. M. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *Proceedings of HPDC 1999*, pages 167–176. IEEE.

[3] C. C. Aggarwal. *Data Streams: Models and Algorithms (Advances in Database Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[4] R. Ammendola, M. Bernaschi, A. Biagioni, M. Bisson, M. Fatica, O. Frezza, F. Lo Cicero, A. Lonardo, E. Mastrostefano, P. S. Paolucci, et al. Gpu peer-to-peer techniques applied to a cluster interconnect. In *IPDPSW 2013*, pages 806–815.

[5] G. Andrade, G. Ramos, D. Madeira, R. Sachetto, R. Ferreira, and L. Rocha. G-dbscan: A gpu accelerated algorithm for density-based clustering. *Procedia Computer Science*, 18:369–378, 2013.

[6] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with datacutter. *Parallel Computing*, 27(11):1457–1478, 2001.

[7] J. Cabezas, M. Jordà, I. Gelado, N. Navarro, and W.-m. Hwu. Gpu-sm: shared memory multi-gpu programming. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, pages 13–24. ACM, 2015.

[8] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998.

[9] T. L. A. de Souza Ramos, R. S. Oliveira, A. P. De Carvalho, R. A. C. Ferreira, and W. Meira Jr. Watershed: A high performance distributed stream processing system. In *SBAC-PAD 2011*.

[10] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and Lumsdaine. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*.

[11] D. Melo, S. Toledo, F. Mourão, R. Sachetto, G. Andrade, R. Ferreira, S. Parthasarathy, and L. Rocha. Hierarchical density-based clustering based on gpu accelerated data indexing strategy. International Conference on Computational Science, ICCS 2016.

[12] C. Nvidia. Programming guide, 2008.

[13] L. Rocha, G. Ramos, R. Chaves, R. Sachetto, D. Madeira, F. Viegas, G. Andrade, S. Daniel, M. Gonçalves, and R. Ferreira. G-knn: an efficient document classification algorithm for sparse datasets on gpus using knn. In *Proceedings of ACM SAC 2015*.

[14] G. Shainer, A. Ayoub, P. Lui, T. Liu, M. Kagan, C. R. Trott, G. Scantlen, and P. S. Crozier. The development of mellanox/nvidia gpudirect over infiniband—a new model for gpu to gpu communications. *Computer Science-Research and Development*, 26(3-4):267–273, 2011.

[15] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[16] K. Tang, Y. Yu, Y. Wang, Y. Zhou, and H. Guo. Ema: Turning multiple address spaces transparent to cuda programming. In *2012 Seventh ChinaGrid Annual Conference*, pages 170–175. IEEE, 2012.

[17] F. Viegas, G. Andrade, J. Almeida, R. Ferreira, M. Goncalves, G. Ramos, and L. Rocha. Gpu-nb: A fast cuda-based implementation of naive bayes. In *SBAC-PAD 2013*, pages 168–175.

[18] S. Wienke, P. Springer, C. Terboven, and D. an Mey. Openacc: First experiences with real-world applications. In *Proceedings of Euro-Par'12*, pages 859–870. Springer-Verlag, 2012.

[19] W. Wu, G. Bosilca, R. vandeVaart, S. Jeaugey, and J. Dongarra. Gpu-aware non-contiguous data movement in open mpi. In *Proceedings of the ACM HPDC 2016*, pages 231–242.

[20] J. Young, S. H. Shon, S. Yalamanchili, A. Merritt, K. Schwan, and H. Fröning. Oncilla: a gas runtime for efficient resource allocation and data movement in accelerated clusters. In *IEEE CLUSTER, 2013*, pages 1–8.