Sparse Matrix Computations on Clusters with GPGPUs

Valeria Cardellini, Alessandro Fanfarillo, Salvatore Filippone
Dipartimento di Ingegneria Civile e Ingegneria Informatica
Università di Roma "Tor Vergata", Roma, Italy
cardellini@ing.uniroma2.it, fanfarillo@ing.uniroma2.it, salvatore.filippone@uniroma2.it

***Abstract*—Hybrid nodes containing GPUs are rapidly becoming the norm in parallel machines. We have conducted some experiments regarding how to plug GPU-enabled computational kernels into PSBLAS, a MPI-based library specifically geared towards sparse matrix computations. In this paper, we present our findings on which strategies are more promising in the quest for the optimal compromise among raw performance, speedup, software maintainability, and extensibility. We consider several solutions to implement the data exchange with the GPU focusing on the data access and transfer, and present an experimental evaluation for a cluster system with up to two GPUs per node. In particular, we compare the pinned memory and the OpenMPI approaches, which are the two most used alternatives for multi-GPU communication in a cluster environment. We find that OpenMPI turns out to be the best solution for large data transfers, while the pinned memory approach is still a good solution for small transfers between GPUs.**

***Keywords*—Sparse matrices, GPGPU computing, Message Passing Interface (MPI)**

I. Introduction

The performance potential of Graphics Processing Units (GPUs) leads to a fast growing interest in using GPUs for General Purpose computing (GPGPU). Hybrid nodes containing GPUs are rapidly becoming the norm in most advanced clusters [1]–[3], and are even being offered as an infrastructure service in Cloud computing (e.g., Amazon EC2 GPU instances).

In scientific applications, many physical problems modeled by partial differential equations (PDEs) are solved via discretizations that transform the original equations into a linear system and/or an eigenvalue problem with a sparse coefficient matrix. A matrix is sparse when most of its elements are zero; this fact is exploited in devising a representation that does not store explicitly the null coefficients. In many applications, such a scheme pays off nicely and sparse matrices are widely used in scientific computations. Therefore, the Sparse Matrix-Vector product (SpMV) is one of the key computational kernels (the so called "Seven Dwarfs") and holds a fundamental role in many scientific and engineering applications [4].

SpMV on GPUs presents new challenges, because many optimization techniques used in general-purpose architectures cannot be directly applied on them. Moreover, sparse matrix structures introduce additional challenges with respect to their dense counterparts, because operations on them are typically much less regular in their access patterns. Therefore, in recent years a significant number of research efforts has been devoted to the efficient implementation of sparse matrix-vector multiplication on a single GPU, among them [5]–[10], and the NVIDIA's CUSP [11] and cuSPARSE [12] libraries, with the aim of optimizing memory pattern efficiency, memory footprint, and fine-grain parallelism.

Parallel Sparse BLAS (PSBLAS)[a] is a library of Basic Linear Algebra Subroutines for parallel sparse applications supporting complex computations on multicomputers [13] and using MPI for inter-process communication. A Fortran 2003 version of PSBLAS has been recently released and forms the basis for the experimental results herein presented.

In this paper we discuss how to integrate GPU-enabled computational kernels for SpMV into the PSBLAS library. We present our findings in the quest for the optimal compromise between raw performance, speedup, software maintainability, and extensibility. We consider various alternative strategies to realize the data exchange needed by a fully parallel version of the PSBLAS library with CUDA support, where multiple PSBLAS processes, each one using a GPU device for the SpMV kernel computation, communicate among them. Specifically, the alternative strategies for data transfers from CPU to GPU and vice versa include CUDA Peer-to-Peer, synchronization, scatter and gather kernels, and static index in two versions, namely standard and pinned memory. We also present a strategy which exploits specialized data transfer support available in OpenMPI. We compare the performance of the various data exchange approaches when executing the sparse matrix-vector multiplication in a heterogeneous cluster environment with different configuration scenarios in terms of number of nodes and number of GPUs. In particular, our experimental results demonstrate that OpenMPI turns out to be the best solution for large data transfers when using multi-GPU communication in a cluster environment, while the pinned memory approach is still a good solution for small transfers between GPUs.

Most research efforts that propose application optimizations on heterogeneous systems with GPUs (e.g. [2], [8], [14]) typically rely on the overlapping of the MPI communication,

[a] http://www.ce.uniroma2.it/psblas

the CPU-GPU communication, and the CPU-GPU computation. In particular, in [8] Kreutzer et al. focused on sparse matrix-vector multiplication on GPGPU clusters and considered three alternatives for communication and computation: no overlap of communication and computation, naive overlap of communication and computation by nonblocking MPI, and the use of dedicated host threads for asynchronous MPI communication. The latter approach achieves the best results in their experimental setting, which however differ from ours both in terms of GPU cluster architecture and set of matrices and therefore the performance results are not comparable.

The rest of the paper is organized as follows. In Section II we provide some background on PSBLAS, GPGPU, and CUDA. In Section III we describe our efforts in developing a serial version of the PSBLAS library with GPU computation support, which has been presented in a previous work [15]. In Section IV we discuss how to enable the MPI communication layer to produce a fully parallel version of PSBLAS with CUDA support. In Section V we analyze alternative solutions to realize the needed data exchange and compare their performance in Section VI through a set of experiments conducted on two different platforms. Finally, we conclude in Section VII and give hints for future work.

II. Background Software

The PSBLAS library has been developed to facilitate the parallelization of applications using iterative solvers for sparse linear systems at the heart of many scientific applications, through the distributed memory paradigm [13], [16], [17]. The current version 3 of the library is implemented in Fortran 2003; its object-oriented model gives effective usability, maintainability and extensibility while at the same time maintaining a good level of performance. PSBLAS employs a distributed memory paradigm based on MPI; every process involved in the computation will produce some data and will share some results with its "neighbours" in order to complete the whole computation.

General Purpose computing on GPUs (or GPGPU) deals with usage of GPUs for purposes different from more conventional graphics applications and has recently seen a dramatic increase in popularity. CUDA is a parallel computing platform and programming model proposed by NVIDIA for its hardware products.

GPUs are throughput-oriented architectures; therefore, they represent an efficient approach to deal with problems characterized by a highly parallelizable solution. PSBLAS is geared towards implementation of iterative solvers; hence we are interested in the performance of the sparse matrix-vector product, since it is the crucial kernel in the given context of scientific and engineering applications.

A key component of CUDA is the GPU memory hierarchy, which contains various levels that differ by speed, size, and scope. They comprise registers, local memory, shared memory, constant memory, texture and surface memory, global memory, and mapped memory. The shared memory has a size of 16 KB or 48 KB for each SM and is arranged in 32 banks, each of which 32 bits wide. A programmer can use shared memory by applying the qualifier `__shared__` to a variable declaration; shared memory is useful because it attains a good compromise between size and access speed. These characteristics have made shared memory widely used in tiles techniques, which allow to increase the performance by recycling the data used in the threads block. The global memory is the largest and slowest memory on a GPU and can be accessed from both device and host; the mapped memory exploits the direct mapping access on (locked) host memory.

The most important problem in using GPUs on sparse matrix computations is the large overhead imposed by the PCIe bus which connects the CPU to the GPU, whose bandwidth typically becomes the performance bottleneck. In fact, for each iteration of our iterative solver, every process will exchange data with its neighbors in order to complete the computation. This means that, for every iteration a transfer from GPU to CPU is required in order to proceed to the next step. When performing a copy from host to GPU, the CUDA driver uses direct memory access (DMA). This operation causes a double copy: the first from the pageable system buffer to a temporary page-locked buffer, and the second from the page-locked buffer to the GPU. Thus, the copy speed is bounded by the slowest between the PCI-E and the system front-side bus. Furthermore, a pageable memory copy involves the CPU, adding further overhead. CUDA provides special functions to allocate host-locked memory, also called *pinned memory*: the operating system guarantees that it will not be paged out [18]. A copy with pinned memory does not need the double access step; moreover, it enables direct use of the host memory inside the CUDA kernels, a working mode called *zero-copy*.

III. PSBLAS with NVIDIA GPUs Support

The latest release of the PSBLAS library (version 3.1.2) is a reimplementation in the Fortran 2003 language; the new internals have a full object-oriented (OO) design as described in [17]. The availability of the OO infrastructure enables an easy implementation of a CUDA "serial" plugin [15]; here "serial" refers to the use of just one PSBLAS process invoking kernels on a single GPU. In this mode, we can test how performant the computational kernels are without the burden of communication among processes.

spGPU[b] is a set of custom matrix storages and CUDA kernels for sparse linear algebra computing on GPU we implemented and includes a new GPU-friendly storage format, named ELL-G [15]. It is a variation of the standard ELLPACK (or ELL) format [19] and aims at reducing the memory overhead of the padding zeros that occurs in ELL. Indeed, the ELL format introduces padding with zero coefficients to fill unused locations of the elements array, but its efficiency highly depends on the distribution of nonzero elements. When

[b] http://code.google.com/p/spgpu/

the number of nonzeros per row varies considerably, the ELL performance degrades due to the overhead of the padding zeros. Since in the GPU implementation it is not necessary to execute arithmetic operations on these padding entries, a better solution is to create an additional array of row lengths, so that each thread will only execute on the actual number of nonzero coefficients within the row, at the cost of one more memory access. Other researchers have used a similar solution, for instance in the ELLPACK-R format described in [10].

The PSBLAS object model enables an easy translation among different sparse matrix formats [15] as well as easy extensibility in user code; it is thus possible to wrap the ELL-G format and use it in the PSBLAS context with the following class (in the code, we refer to ELL-G as `elg` because PSBLAS matrix format names are 3 characters long):

```
type, extends(psb_d_ell_sparse_mat) &
     & :: psb_d_elg_sparse_mat
#ifdef HAVE_GPU
  type(c_ptr) :: deviceMat = c_null_ptr
contains
  procedure, pass(a) :: &
     & d_sp_mv => psb_d_elg_vect_mv
  procedure, pass(a) :: &
     & d_csmm => psb_d_elg_csmm
  generic, public     :: &
     & cp_from => psb_d_elg_cp_from
  procedure, pass(a) :: psb_d_elg_mv_from
  generic, public     :: &
     & mv_from => psb_d_elg_mv_from
  procedure, pass(a) :: &
     &to_gpu  => psb_d_elg_to_gpu
#endif
end type psb_d_elg_sparse_mat
```

The `deviceMat` attribute holds a pointer to a shadow image of the matrix data structure which resides in the GPU memory space. A similar encapsulation is applied for vectors.

A. spGPU

spGPU is a library which implements the computational kernels in CUDA C language as well as the data movement operators invoked from PSBLAS when dealing with the shadow-memory copies of matrices and vectors. Once the data is in the GPU memory, it is possible to invoke the computational kernel, such as that for the sparse matrix-vector product $y \leftarrow \alpha Ax + \beta y$ shown below:

```
__global__ void Dspmvm_gpu_krn (double *y,
     double alpha, double* cM, int* rP, int* rS,
     int n, int pitch, double *x,
     double beta, int firstIndex) {
  int i = threadIdx.x + blockIdx.x * (THREAD_BLOCK);
  if (i >= n)  return;
  double y_prod = 0.0;  int row_size = rS[i];
  rP += i;     cM += i;
  for (int j = 0; j < row_size; j++){
    int pointer = rP[0] - firstIndex;
    double value = cM[0];
    rP += pitch; cM += pitch;
    y_prod += __dmul_rn(value, x[pointer]);
  }
  if (beta == 0.0)
    y[i] = (alpha * y_prod);
  else
    y[i] = __dmul_rn(beta, y[i]) +
          __dmul_rn(alpha, y_prod);
}
```

In the matrix-vector product code above, each row of the product is dispatched to a separate CUDA thread.

IV. From Serial to Parallel Version

The integration of spGPU with PSBLAS as so far described provides a working serial version of PSBLAS with NVIDIA GPU support; its main design issues and some performance results have been discussed in [15], [20].

To produce a fully parallel version of PSBLAS with CUDA support (which we refer to as **PSBLAS-GPU** in the following), we need to enable the MPI communication layer among the various PSBLAS processes, each employing a GPU for the serial part of the computation. The essential communication step is a "halo exchange" [13]. Every sparse matrix can be viewed as a graph representation; for matrices arising from the PDE discretization, this graph has a natural isomorphism with that describing the topology of the discretized computational domain. When a domain is partitioned into subdomains, each one assigned to a process, the nodes of the graph lying at the boundary of a subdomain are involved in data exchange with the adjacent nodes lying just across the boundary; those adjacent nodes from other subdomains are the "halo" of a given domain, and they correspond to the data items to be exchanged. Note that in a normal situation, the number of boundary nodes will be much smaller than the total number of nodes in the subdomain, i.e., we have a surface-to-volume effect. To perform the data exchange, each process has to loop through the set of all adjacent subdomains and for each subdomain it has to collect and send the values at the boundary nodes, as well as to receive from the other processes the values corresponding to the halo nodes.

Therefore, each communication phase has a packing step, a network send, a network receive, and an unpacking step. The packing and unpacking steps, which we call *gather* and *scatter*, are quite poor in terms of coalesced memory accesses on GPU due to their irregular access pattern.

- A gather operation packs various elements from a source vector into a contiguous target vector:
  ```
  for(i=0;i<n;i++)
     y[i] = x[index[i]];
  ```
  This is a typical operation that is used in PSBLAS to prepare a buffer `y` to be sent in the data exchange that is inherent in the parallel sparse matrix-vector product.
- The scatter operation is the inverse of the gather one: we use a received buffer to update a set of vector elements in predefined locations as in the example below:
  ```
  for(i=0;i<n;i++)
  ```

```
x[index[i]] = y[i] +
    beta*x[index[i]];
```

The main problem is the bandwidth bottleneck in moving data between CPU and GPU; the irregular access pattern worsens the situation since it precludes an effective use of coalescent accesses; if all data resides in global memory we are in the worst-case scenario. Some help comes from the L2 cache in the NVIDIA's Fermi and Tesla architectures, which aims to mitigate the effect of irregular memory accesses and can bring up to one order of magnitude of performance improvement when the random accesses are localized by sorting. In any case, it is essential to somehow minimize the amount of required data transfers.

V. Parallel PSBLAS-GPU alternatives

In this section we analyze the possible approaches to implement the data exchange needed for PSBLAS-GPU, highlighting advantages and drawbacks of each alternative.

A. CUDA Peer-to-Peer

Unified Virtual Address (UVA) [21] is a new feature introduced from CUDA 4.1. It allows CPU and GPU to see the same Virtual Address Space; this means that, within a process, the CPU can access/manage the memory allocated on every GPU installed on the node. Another improvement brought by CUDA 4.1 is the Peer-to-Peer access. This mechanism allows two GPUs, installed on the same node, to communicate directly without requiring the CPU's intervention.

Since our programming model uses different processes, Peer-to-Peer cannot be directly used. In our software architecture, the CPU acts as a front-end for the various computation and we use it to control the scatter/gather operations. Thus, the usage of multiple GPUs with Peer-to-Peer access would require a specialized storage format that extends across multiple GPUs, which we have not implemented so far.

B. Sync: Brute Force Solution

The simplest solution is to use a `sync` operation to move the vector data between device and host for each scatter/-gather operation. The `sync` method is normally intended to seed the device version upon start of a computation, and to recover the results at the end of a (possibly long) chain of operations carried out within the GPU. Since the parallel halo communication is handled by existing CPU-side code, this implementation does not require any specific GPU code beyond the serial data movement. It is clear that such solution is not optimal, since at each step we will be moving around a much larger set of data than necessary; therefore, this strategy will only establish a minimum baseline performance to be improved upon.

C. Scatter and Gather Kernels

A better solution is to implement the gather/scatter methods inside the GPU, so as to only transfer the boundary data between host and device. This solution is fairly simple but presents two drawbacks: the irregularity of the memory access patterns and the need to handle arrays of indices. Indeed, the indirect addressing is based on a data structure that is built on the host side. Therefore, to execute the gather/scatter operations we need to have a copy of the indices on the device side, and this requires the invocation of the `cudaMalloc()`, `cudaMemcpy()`, and `cudaFree()` kernels.

D. Pinned Memory Version

To avoid the data traffic associated with the indices and the buffers needed for packing/unpacking, one possible strategy is to use the mapped memory. The latter is a particular page-lock host memory allocation provided by CUDA, which can be accessed directly by CUDA kernels.

Using the mapped memory for the indices and the source/destination buffers we can avoid the time spent on allocating, copying, and deallocating; furthermore, there is a significant improvement of the PCI-E bus utilization. CUDA provides a memory management function called `cudaHostRegister()` which transforms a normal host memory area into a pinned memory area; previous restrictions on its usage have been lifted from CUDA 4.1. On the library side, the use of pinned memory requires to store a couple of new fields in the data structures and to register/deregister them as needed during the application lifetime.

E. Static Index on GPU (Standard and Pinned Version)

As already seen, a well-known best practice for optimizing codes on NVIDIA GPUs and other accelerators is "send data on GPU and keep it there".Since the index lists in the communication are based on the topology of the discretization mesh, they are quite stable during the application life; the lifetime of a communication descriptor is tied to the lifetime of a discretization mesh, and certainly this covers multiple matrix-vector products, i.e., data exchanges. It is therefore clear that a viable solution would be to keep a copy of the indices in the device memory throughout the life of the descriptor object. An alternative is to keep the indices in pinned memory. In the rest of the paper, we will refer to the two alternatives as *IndexStandard* and *IndexPinned*, respectively.

F. Open MPI with CUDA Support

A completely different solution can be implemented through the use of MPI derived data types, specifically using the `MPI_Type_indexed` function, which allows the creation of a data type with irregular stride(s) among the components. We can then use a specific derived data type to describe the boundary elements taking part in a data exchange. The upshot

would be that the packing and unpacking operations would be performed directly by the MPI library.

This approach becomes interesting when coupled with the support provided by Open MPI [22] for the UVA usage [23], so that all pointers within a program have unique addresses, and a new API that allows to check if a pointer is either a CUDA device pointer or a host memory pointer (used by the library to detect if the memory area used in a send/receive is a device area or not). Furthermore, CUDA 4.1 adds the CUDA IPC (InterProcess Communication), which allows a fast communication between GPUs on the same node, also between different processes. In addition, CUDA 4.1 provides the ability to register host memory with the CUDA driver which can improve performance.

In other words, it is possible to delegate *both* packing and movement between host and device memory to the MPI implementation. At the time of this writing, the OpenMPI CUDA-aware support is approaching full maturity. It now fully exploits the capabilities of the newer CUDA versions, such as GPUDirect RDMA available from CUDA 6.0 in Kepler-class GPUs, and is quite stable.

VI. Experimental Results

The experimental results presented in this section are organized in two different sets. We first analyze in Section VI-A a comparison among the "CUDA native" approaches, which include: Sync, Scatter/Gather (for short, SG), Pinned, and the two variants of Static Index, namely IndexPinned and IndexStandard, which have been described from Section V-B to Section V-E. Then, in Section VI-B we present a comparison between the best solution coming from the CUDA native comparison and the OpenMPI with CUDA support approach.

The CUDA native experiments have been executed on the Jazz GPU cluster provided by CINECA[c]; its nodes are based on dual-socket esa-core Intel Xeon X5650 processors (for a total of 12 cores), with 48 GB RAM; each node has two NVIDIA Fermi S2050 devices and is interconnected with a QDR InfiniBand (40 Gbit/s). The second set of experiments has been executed on Amazon Web Service (AWS) EC2 GPU instances of type CG1; each node is equipped with 2 x Intel Xeon X5570, quad-core with hyperthread plus 2 NVIDIA Tesla M2050 GPUs. Each AWS CG1 instance is interconnected with a low latency 10 Gbit/s network. Furthermore, a scalability test has been executed on several AWS EC2 G2 instances; each node has an Intel Xeon E5-2670 (Sandy Bridge), 15GB RAM plus one NVIDIA GRID Kepler GK104. Note that both CG1 and G2 instances do not not have an Infiniband network.

To evaluate the PSBLAS-GPU performance, we used a relatively simple main program which iterates the sparse matrix-vector multiplication for a specified number of times; the performance metric of interest is the global throughput expressed in GFLOPS. The sparse matrix is generated from an advection-diffusion equation on the unit cube. The equation is discretized with a simple centered differences strategy, giving rise to a matrix with at most 7 nonzeros per row: the matrix size is expressed in terms of the length of the cube edge, so that the case pde10 corresponds to a 1000×1000 matrix. We already used this sparse matrix collection in [15].

The availability of two GPUs per node enables different configurations with the same number of processes. However, switching from single to dual occupancy per node is by no means neutral in terms of performance, because with double occupancy, contention for the PCI-E bus becomes the main performance bottleneck.

A. CUDA Native Experiments

During a preliminary testing phase, the Sync approach obtained, as expected, the lowest throughput; therefore, it has been switched off at large memory sizes.

1) 1 Node with 2 GPUs (1-2): With 1 node and 2 GPUs (for brevity, 1-2 scenario) the interconnection network is not involved; Figure 1(a) shows the corresponding performance comparison.

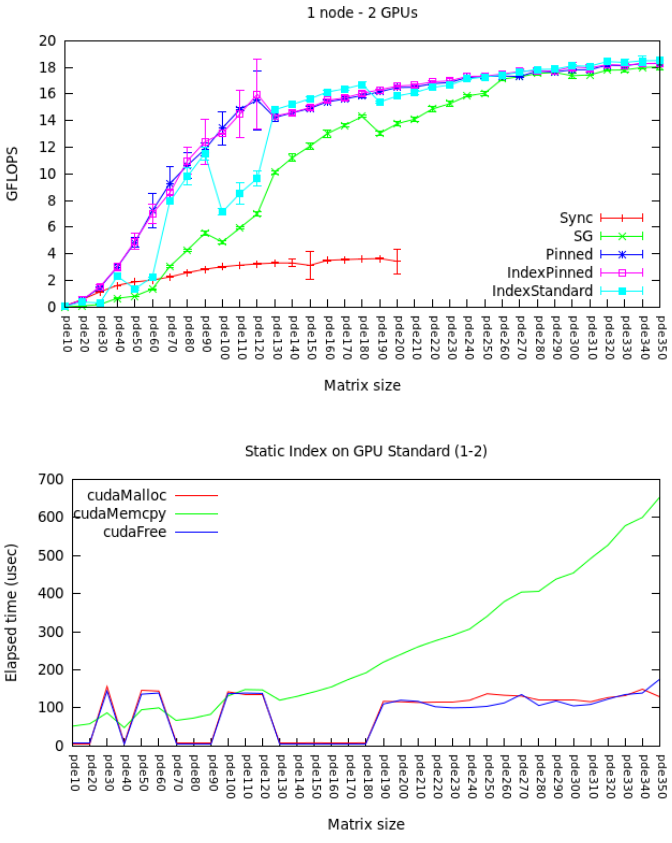

Figure 1. Throughput (a) and elapsed time in CUDA functions (b) for the (1-2) scenario on CINECA platform

The poor performance obtained by the Sync approach is evident and expected because of the involved memory bottle-

[c] http://www.cineca.it

neck. The performance dips exhibited by the Scatter/Gather (SG) and IndexStandard approaches are slightly surprising; they have been found to be reproducible and are ultimately caused by the behavior of the memory management routines `cudaMalloc()`, `cudaMemcpy()` and `cudaFree()`. The SG approach uses them on both indices array and MPI buffer, while indexStandard uses them only on the MPI buffer; thus, IndexStandard has a similar trend to SG but better performance. This explanation is confirmed by the pinned alternatives, showing a much smoother performance curve.

To confirm our explanations we collected detailed timings with the same number of repetitions as in the matrix-vector case; Figure 1(b) shows the results, clearly indicating that the memory management overhead has a rather complex behavior, probably related to internal algorithmic and/or hardware thresholds. The sharp rises and falls in Figure 1(b) match nicely those reported in Figure 1(a).

2) 2 Nodes with 1 GPU (2-1): In the 2 nodes with 1 GPU scenario (for brevity, 2-1) we use only one core and one GPU from each node; the results shown in Figure 2(a) indicate a better performance obtained by all the approaches with respect to the 1-2 scenario, because of the absence of contention on the PCI-E bus.

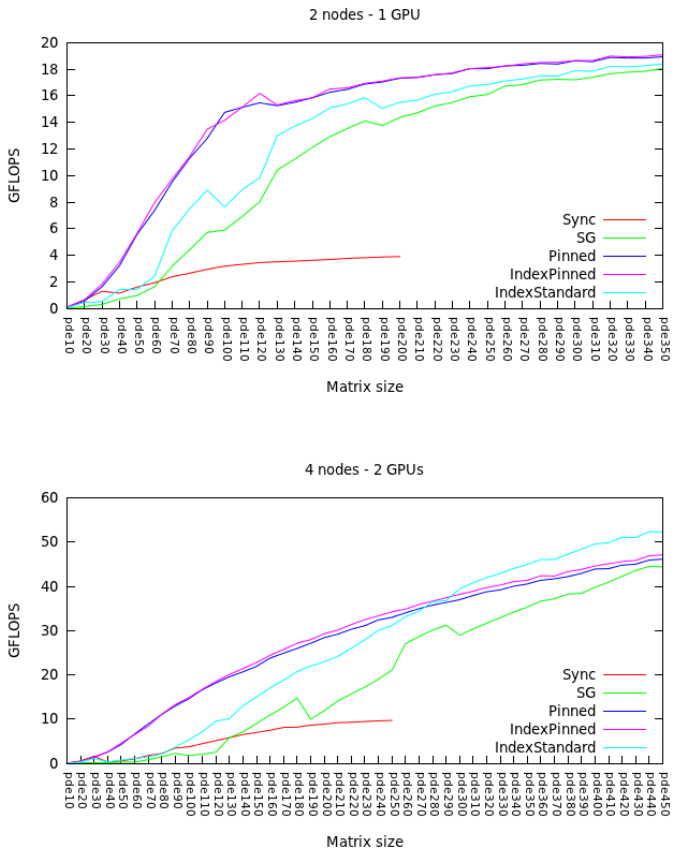

Figure 2. Throughput for 2-1 scenario and 4-2 scenario on CINECA platform

This is substantiated by a simple memory bandwidth test, measuring the time needed to transfer a fixed amount of data from CPU to GPU for both pageable and pinned memory. Using 1 node with 2 GPUs and pageable memory we reach 3034.9287 MB/s, whereas using pinned memory we reach 3614.0933 MB/s. When we employ 2 nodes with 1 GPU, with pageable memory we obtain a throughput of 3457.4 MB/s, and with pinned memory we reach 5510.7 MB/s.

In the larger scenario with 4 nodes and 2 GPUs per node of Figure 2(b), we see that the IndexStandard approach overtakes the Pinned and IndexPinned ones for matrix sizes over pde300; as in the 1-2 case, the pinned alternatives suffer from contention on the bus.

B. CUDA Native vs. MPI-Based Experiments

For the second set of experiments executed on the AWS EC2 GPU cluster, we select the IndexPinned approach as the best performing one from the previous comparison and we name it as Pinned in the corresponding performance curves.

1) 1 Node with 2 GPUs MPI (1-2): With 1 node and 2 GPUs the results reported in Figure 3 show that the OpenMPI support for CUDA IPC works pretty well for sparse matrices having more than 1 million rows.

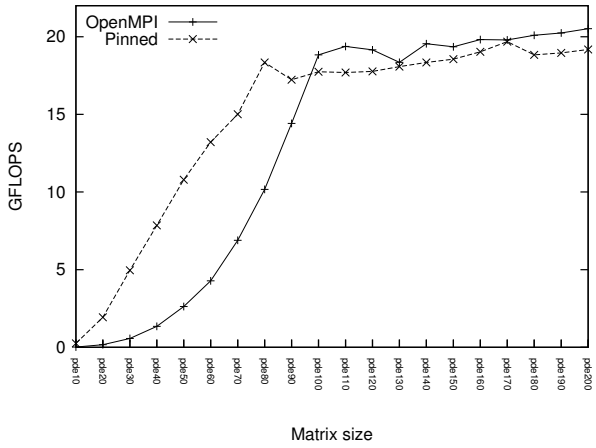

Figure 3. Throughput for (1-2) scenario on AWS platform

2) 2 Nodes with 1 GPU MPI (2-1): In the 2 nodes with 1 GPU scenario, we have to consider the impact of the network overhead. Since we use two nodes interconnected by a 10 Gbit/s network, as shown in Figure 4 we see an expected performance decrease when compared to the 1-2 scenario reported in Figure 3. Also in this scenario the OpenMPI-based implementation works well for matrix sizes larger than 1 million rows.

3) 2 Nodes with 2 GPUs MPI (2-2): In the 2 nodes with 2 GPUs setting shown in Figure 5, we found that the OpenMPI approach shows great instability with respect to the Pinned one. There are two possible motivations for such a strange behavior. The first one is related to the data partition algorithm: for every test we use a block partition algorithm which

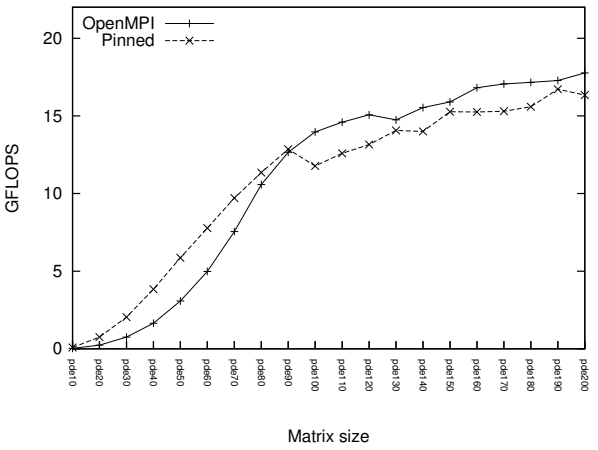

Figure 4. Throughput for (2-1) scenario on AWS platform

operates in a "data blind" way. The quick jumps between high and low performance appear to be related with load imbalance. Indeed every matrix with a size non divisible for 4 (number of processes in the scenario) is affected by low performance. During the scalability test executed on the G2 instances we observed the same unstable behavior. This means that the CUDA IPC are not directly related with such instability (CUDA IPC is used only on multi-GPU nodes). However, the same partition algorithm has been applied to the Pinned approach which does not show the unstable behavior; thus we believe that the OpenMPI CUDA support is more sensitive to load imbalance, although this will require further investigation.

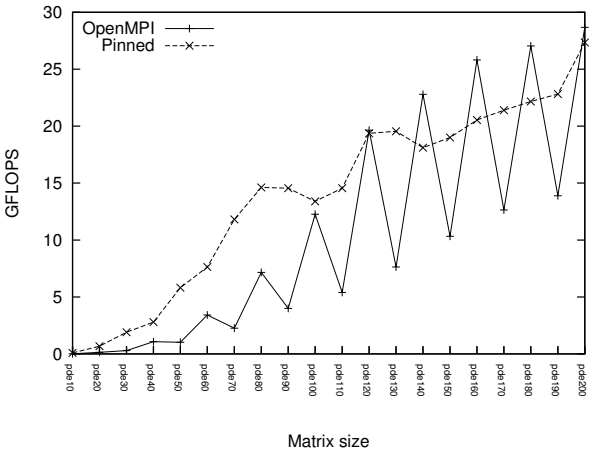

Figure 5. Throughput in (2-2) scenario on AWS platform

In order to figure out where the instability comes, we applied the TAU Performance System on our application. For matrices with size not exactly divisible for the number of processes we pay a penalty during the `MPI_Send()` execution.

4) Scalability Test: In order to find out which data exchange approach achieves the best scalability performance, we ran a scalability test on up to 8 EC2 G2 instances, where each instance has only one GPU. In Figure 6 the y-axis represents the weak scalability efficiency. As regards the matrices used for the scalability test, for the single node execution we used the pde160 matrix and we doubled the memory occupation every time we doubled the number of nodes.

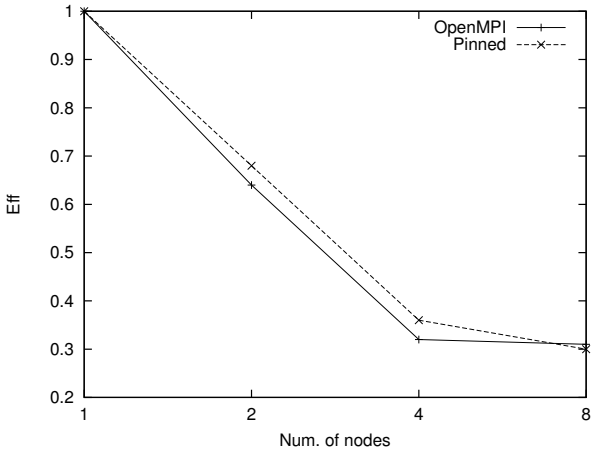

Figure 6. Scalability on AWS platform using EC2 G2 instances

Figure 6 shows that without any support for the OpenMPI features the IndexPinned approach performs better than the OpenMPI one. This is reasonable because OpenMPI adds the overhead needed to "understand" which kind of memory (CPU or GPU) it handles.

VII. Conclusions

In this paper, we have analyzed how to integrate GPU-enabled computational kernels into PSBLAS and we have considered several solutions to implement the data exchange with the GPUs. In particular, we have presented a comparison between the two most used alternatives for multi-GPU communication in a cluster environment. The pinned memory approach is still a good solution for small transfers between GPUs. The great improvement of OpenMPI compared to a couple of years ago brings an unexpected performance enhancement: it turns out to be the best solution for large data transfers in every experiment. However, on a bare cluster without any support for the OpenMPI features, we have that the Pinned version performs better than the OpenMPI version.

We also tested the alternative data exchange approaches using a different set of sparse matrices taken by the Tim Davis's collection. The results do not differ from those shown and therefore for space reason we have not reported them.

In future work we plan to implement a data aware partition algorithm in order to optimize the load balance among the processes. Furthermore, we plan to test a version which uses the GPUDirect support provided by OpenMPI. We will

also consider the recently proposed GPU-Aware MPI [24], which supports data communication from GPU to GPU using standard MPI and has been incorporated into MVAPICH2. We have already run some preliminary tests about MVAPICH2 with CUDA-aware support and the results are very promising.

ACKNOWLEDGMENT

We gratefully acknowledge the support we have received from CINECA for project IsC14_HyPSBLAS, under the ISCRA grant programme for 2014, and from Amazon with the AWS in Education Grant programme 2014.

REFERENCES

[1] V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, and W.-M. Hwu, "GPU clusters for high-performance computing," in *Proc. of IEEE Int'l Conf. on Cluster Computing and Workshops*, ser. CLUSTER '09, Aug. 2009, pp. 1–8.

[2] D. Jacobsen, J. Thibault, and I. Senocak, "An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters," in *48th AIAA Aerospace Sciences Meeting*, 2010.

[3] C.-T. Yang, C.-L. Huang, and C.-F. Lin, "Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters," *Computer Physics Communications*, vol. 182, no. 1, pp. 266–269, 2011.

[4] P. Colella, "Defining Software Requirements for Scientific Computing," 2004, http://view.eecs.berkeley.edu/w/images/temp/6/6e/20061003235551!DARPAHPCS.ppt.

[5] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. of Int'l Conf. on High Performance Computing Networking, Storage and Analysis*, 2009.

[6] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," *SIGPLAN Not.*, vol. 45, pp. 115–126, Jan. 2010.

[7] H.-V. Dang and B. Schmidt, "CUDA-enabled sparse matrix-vector multiplication on GPUs using atomic operations," *Parallel Computing*, vol. 39, no. 11, pp. 737–750, 2013.

[8] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. R. Bishop, "Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation," in *Proc. of 26th IEEE Int'l Parallel and Distributed Processing Symposium Workshops & PhD Forum*, ser. IPDPSW '12, 2012, pp. 1696–1702.

[9] M. Maggioni and T. Berger-Wolf, "AdELL: An adaptive warp-balancing ELL format for efficient sparse matrix-vector multiplication on GPUs," in *Proc. of 42nd IEEE Int'l Conf. on Parallel Processing*, ser. ICPP '13, Oct. 2013, pp. 11–20.

[10] F. Vazquez, G. Ortega, J. J. Fernández, and E. M. Garzon, "Improving the performance of the sparse matrix vector product with GPUs," in *Proc. of 10th IEEE Int'l Conf. on Computer and Information Technology*, ser. CIT '10, 2010, pp. 1146–1151.

[11] NVIDIA Corp., "CUDA CUSP library," 2014, http://developer.nvidia.com/cusp.

[12] ——, "CUDA cuSPARSE library," 2014, http://developer.nvidia.com/cusparse.

[13] S. Filippone and M. Colajanni, "PSBLAS: a library for parallel linear algebra computations on sparse matrices," *ACM Trans. on Math Software*, vol. 26, pp. 527–550, 2000.

[14] M. Bernaschi, M. Bisson, T. Endo, S. Matsuoka, M. Fatica, and S. Melchionna, "Petaflop biofluidics simulations on a two million-core system," in *Proc. of 2011 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2011.

[15] V. Cardellini, S. Filippone, and D. Rouson, "Design patterns for sparse-matrix computations on hybrid CPU/GPU platforms," *Scientific Programming*, vol. 22, no. 1, pp. 1–19, 2014.

[16] S. Filippone and A. Buttari, "PSBLAS 3.0-beta user's guide: a reference guide for the Parallel Sparse BLAS library," Jun. 2013, http://www.ce.uniroma2.it/psblas/psblas-3.1.pdf.

[17] ——, "An object model for sparse matrix computations in Fortran 2003," *ACM Trans. on Math Software*, vol. 38, no. 4, 2012.

[18] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 2nd ed. Morgan Kaufmann, 2012.

[19] R. Grimes, D. Kincaid, and D. Young, "ITPACK 2.0 user's guide," Center for Numerical Analysis, University of Texas, CNA-150, 1979.

[20] D. Barbieri, V. Cardellini, S. Filippone, and D. Rouson, "Design patterns for scientific computation of sparse matrices," in *Europar 2011 Workshops, part I*, ser. LNCS 7155. Springer, 2012, pp. 367–376.

[21] T. Schroeder, "Peer-to-Peer & Unified Virtual Addressing," 2011, http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_GPUDirect_uva.pdf.

[22] "Open MPI: Open Source High Performance Computing," http://www.open-mpi.org/.

[23] "What kind of CUDA support exists in Open MPI?" http://www.open-mpi.org/faq/?category=running#mpi-cuda-support.

[24] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "GPU-aware MPI on RDMA-enabled clusters: design, implementation and evaluation," *IEEE Trans. Parallel Distrib. Syst.*, 2014, To appear.