On the Execution of Computationally Intensive CPU-based Libraries on Remote Accelerators for Increasing Performance: Early Experience with the OpenBLAS and FFTW Libraries

Santiago Mislata and Federico Silla
Universitat Politècnica de València, Spain
Email: sanmisva@gap.upv.es, fsilla@disca.upv.es

***Abstract*—Virtualization techniques have shown to report benefits to data centers and other computing facilities. In this regard, virtual machines not only allow reducing the size of the computing infrastructure while increasing overall resource utilization but virtualizing individual components of computers may also provide significant benefits. This is the case, for example, for the remote GPU virtualization technique, implemented in several frameworks during the last years.**

In this paper we present an initial implementation of a new middleware for the remote virtualization of another component of computers: the CPU itself. Our proposal uses remote accelerators to perform computations that were initially intended to be carried out in the local CPUs, doing so transparently to the application and without having to modify its source code. By making use of the OpenBLAS and FFTW libraries as case studies, we carry out a performance evaluation targeting several system configurations comprising Xeon processors as well as Ethernet and InfiniBand QDR, FDR, and EDR network adapters in addition to NVIDIA Tesla K40 GPUs. Results not only demonstrate that the new middleware is feasible, but they also show that mathematical libraries may experience a significant speed up, despite of having to move data forth and back to/from remote servers.

***Keywords*-Virtualization, GPUs, mathematical libraries**

I. Introduction

Virtualization technologies, such as virtual machines, have demonstrated to provide noticeable economic savings to data centers, the main reason being that they can be concurrently executed in a real computer, thus sharing its resources and therefore increasing overall utilization. This is why solutions such as VMware [1], Xen [2], KVM [3], or VirtualBox [4] are so popular nowadays. Actually, the benefits reported by the use of virtual machines has motivated that leading processor manufacturers such as Intel or AMD have increasingly incorporated more support for virtualization into their chip designs [5]. This has also been the case for network fabric manufacturers. In this regard, not only the ones at the highest performance end, like Mellanox Technologies with the InfiniBand cards [6], but also those for more modest technologies, such as Ethernet [7], have also included virtualization mechanisms into their designs. These mechanisms basically allow replicating, at the logical level, the network card so that each of the replicas can be assigned to one of the virtual machines. In a similar way, NVIDIA has recently included desktop virtualization support within its new GRID K1 graphics processing units (GPUs) so that a given physical GPU can be shared among up to eight virtual machines [8]. Intel has also recently introduced virtualization support for its GPUs within the KVM framework [9]. As can be seen, in order to efficiently support the virtualization of entire computers, it is also necessary to virtualize individual components of the computer.

Nevertheless, virtualizing individual components of the computer that hosts the application making use of them is not the only option to attain economical savings. External resources may also be virtualized and shared among several concurrent users, providing even larger benefits. This is the case of networked disks, which allow a file system to be shared among many different computers. Likewise, it is possible to provide GPU-acceleration services to a cluster by sharing a networked GPU by means of the remote GPU virtualization technique, which has been implemented in frameworks such as rCUDA [10], GVirtuS [11], or DS-CUDA [12], among others. Furthermore, when remote GPU virtualization solutions are considered at the cluster level, they provide noticeable reductions in the total execution time of a given workload composed of a set of computing jobs [13] and in the total energy required to execute such workloads [14].

However, computations on GPUs are not the only ones that might be offloaded to other cluster nodes. In this regard, computations initially intended to be carried out in the CPU cores of a computer might also be offloaded, in a transparent way, to accelerators located at other cluster nodes. In this case, however, in addition to the flexibility provided by the remote virtualization technique, reductions in execution time may be achieved, as it will be shown later.

In this paper we present a first implementation of a new middleware that allows offloading to accelerators located in other nodes of the cluster the computationally intensive CPU parts of an application. This is done without having to modify the source code of applications and in a completely transparent way to them. The rest of the paper is organized as follows. Section II presents the proposal in detail. After that, we present in Section III a performance evaluation based on real executions of the new middleware in several system configurations. Next, Section IV presents a performance estimation of the new middleware when the communication layer is improved. Later, Section V introduces an optimization to the offloading process. Finally, Section VI presents the main conclusions of this work.

II. Offloading CPU Computations to Remote Accelerators

The idea of the new middleware is simple: it moves the computationally intensive parts of an application, written to be executed in the local CPUs, to accelerators installed in other nodes of the cluster, and doing so without modifying the application source code. In this way, the computation to be performed in the local CPU cores by mathematical libraries such as BLAS, LAPACK, or FFT, among others, is transparently offloaded to accelerators in other cluster nodes.

The new middleware is organized following a client-server distributed architecture, as shown in Figure 1. The client side is automatically contacted by the application as soon as it makes a call to one of the functions of the replaced mathematical libraries. Both the application and the client middleware are executed in the same computer. The client side of the new middleware presents to applications the very same interface as the BLAS, FFT, LAPACK, or other libraries do. This is achieved by creating a different wrapper for each of the functions in the mathematical libraries. Upon reception of a request from the application, the client middleware processes it and forwards the appropriate command, along with the data, to the remote server, which interprets the request and performs the required processing by accessing the real accelerator in order to execute the corresponding mathematical function. Once the accelerator has completed the execution of the requested function, results are sent back to the client middleware, which delivers them to the demanding application. Notice that the application is not aware of this process. It made a call to a mathematical function, according to its original source code, and after some time the call is completed and execution resumed, exactly in the same way as if the computation would have been carried out in the local CPU, as it is intended in the unmodified application code.

In order to integrate the new middleware with applications in an automatic and transparent way, the new framework replaces the mathematical libraries by a library containing a set of function wrappers that will be called whenever one of the original mathematical functions is to be executed. These wrappers will take the arguments of the original function and forward the input data of the function to the actual node of the cluster that will perform the requested computation in the accelerator.

Using the new middleware is straightforward. First, the library file containing the set of wrappers at the client side should be copied to the computer executing the application, which should additionally be compiled so that it uses dynamic libraries, which is the common case. Furthermore, several environment variables should be set. In the case for the Linux operating system, for instance, the already existing LD_LIBRARY_PATH environment variable should be set according to the final location of the client

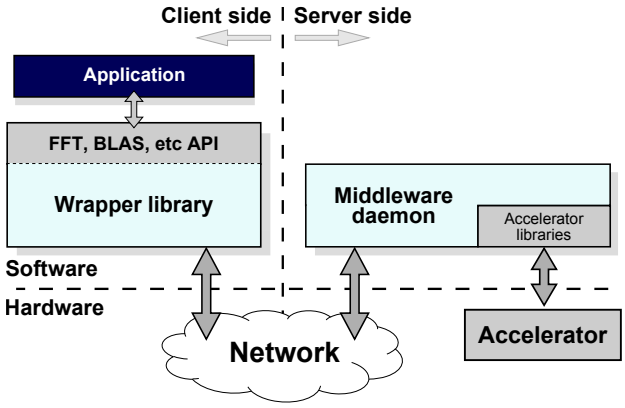


Figure 1: Architecture of the new middleware.

middleware file. After that, a new environment variable called RCPU_SERVER should be initialized with the IP address of the computer hosting the server side of the new middleware and the port number used by the server to receive requests. The syntax for initializing this variable is "RCPU_SERVER=IP_address:port". At the remote node, the binaries of the middleware server should be executed. This server is configured as a daemon and waits for computation requests in a configurable port number. In case other operating systems are used, a similar procedure should be followed.

It is important to remark that the proposed middleware is not limited to certain libraries in the client node neither in the server computer. On the one hand, in the client node any library that performs intensive computations in the CPU is eligible to be remotely accelerated. On the other hand, any library implemented for an accelerator could be used in the remote computer in order to serve computations for the equivalent CPU-based functions (as long as the remote computer includes such accelerator, obviously). Figure 2 depicts this idea. In the client node several CPU-based libraries have been replaced by the wrappers of the proposed middleware. Other libraries not depicted in the figure could also be considered. In the server side several libraries might be used to service client requests. For example, if the server uses the GPU technology in order to provide acceleration, then the cuBLAS [19] library by NVIDIA might be leveraged to accelerate the computations of libraries such as GotoBLAS [22] or OpenBLAS [18]. Similarly, the Magma [21] and Plasma [21] libraries could also be used. It is also possible that the server features several GPUs. In this case the cuBLAS-XT library [20] by NVIDIA could be leveraged in the server to distribute the computations among the available GPUs. Other libraries such as Magma might also be used to take advantage of multi-GPU servers. Notice that distributing data among the GPUs would be transparently done by the middleware server and, therefore,

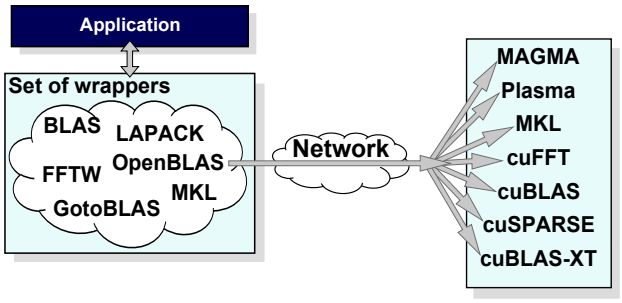


Figure 2: Architecture of the new middleware from a library perspective.

the application will still perform a single call to the CPU-based original function, without requiring any change in its source code. In a similar way, in order to accelerate libraries such as FFTW [23], the NVIDIA cuFFT [24] library might be used in the server. Other libraries implementing the FFT computation in GPUs could also be employed. It would even be possible to use one library or the other according to the exact performance of each of them depending on problem size. In case the remote server includes the Intel Xeon Phi accelerator, the MKL library [25] by Intel could be used to accelerate the computations requested by the client node.

Regarding the initial implementation of the new middleware presented in this paper, it makes use of TCP/IP based communications between clients and servers, although a more sophisticated communication layer based on the InfiniBand Verbs API will be developed in the future. Also, the new middleware currently supports only NVIDIA GPUs, but the use of Intel Xeon Phi accelerators is intended to be supported in future versions. Finally, this initial implementation of the new middleware provides partial support for the BLAS library whereas some support for the FFTW library is also implemented. Support for the LAPACK library will be addressed in future versions.

III. Performance Evaluation

In this section we analyze the benefits that the new middleware might provide to applications. For so, we start with an initial analysis based on three representative functions of the BLAS [15] library: DGEMM, DGEMV, and DDOT. These three functions, which use double precision data in our experiments, present different computational complexity. DGEMM basically performs a matrix-matrix product, denoted by the formula $C = \alpha AB + \beta C$ being A, B, and C matrices whereas α and β are scalars. DGEMM presents a complexity O(n^3). Because of this complexity, it is said that the DGEMM function belongs to the Level 3 subset of BLAS functions. On the other hand, DGEMV performs a matrix-vector product, according to the formula $y = \alpha Ax + \beta y$ being A a matrix, x and y vectors, and finally α and β are scalars. DGEMV presents a complexity O(n^2), thus belonging to the Level 2 subset of BLAS. Finally, the DDOT function performs the dot product of two vectors, given by the formula $x^T y$ being x and y vectors. DDOT has complexity O(n), being classified as a Level 1 function.

Another interesting point of view to analyze these functions is the amount of computations performed per input data element. Assuming the use of square matrices in order to simplify this analysis, the DGEMM function requires $3n^2$ input elements, being n the matrix dimension, and performs in total $n^3 + 2n^2$ multiplications and $n^3 + n^2$ additions. Therefore, the amount of computations per input data element is $(n + 2)/3$ multiplications and $(n + 1)/3$ additions, what might be simplified, given the higher complexity of multiplications, to $(n + 2)/3$ operations. In the case for DGEMV, it performs $n^2 + 2n$ multiplications and $n^2 + n$ additions, involving n^2+2n data elements (n being both the matrix dimension and also the vector length). Thus, making a similar simplification as before, the DGEMV function performs 1 operation per data element. Finally, the DDOT function performs n multiplications and n additions with $2n$ data elements, thus accounting for a total of 0.5 operations per input data element. As can be seen, the different computation complexity of these functions translates into a different amount of computations per data element. In a similar way, it also translates into a different computation-to-communication ratio, which will be used later to explain some of the results obtained in this section.

Regarding the testbed used in this section, both the client and server nodes feature two Intel Xeon hexa-core E5-2620v2 processors (Ivy Bridge) operating at 2.1 GHz and 32 GB of DDR3 memory at 1600 MHz. The server also includes an NVIDIA Tesla K40 comprising 12 GB of GDDR5 memory. The Linux CentOS release 6.4 was used. With respect to the network fabric we have considered InfiniBand QDR and FDR network adapters delivering a maximum theoretical bandwidth, respectively, of 40 and 56 Gbps. Additionally, the widely available 1 Gbps Ethernet fabric was also considered. Furthermore, notice that TCP/IP communications between the client and server nodes are leveraged over Ethernet and InfiniBand. In this regard, no kind of optimization has been implemented in the communication layer of the new framework, which in this initial implementation is quite simple: the client side sends all the input data of the outsourced function to the remote server using the standard TCP socket API and, once all the matrices and vectors involved in the requested computation have arrived at the server, they are copied to the GPU memory using the appropriate CUDA commands. In this way, there is no parallelism nor pipeline in the data movement from main memory in the client node to the GPU memory in the remote server. Actually, this is the worst scenario for the new middleware given that these non-optimized communications cause the biggest possible overhead.

Figure 3 depicts the performance attained by the DGEMM

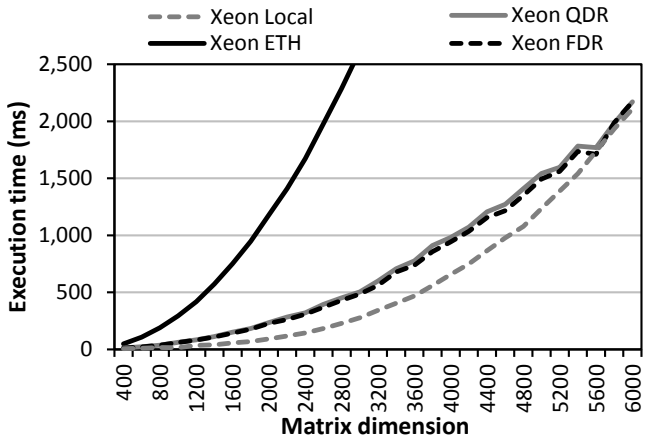
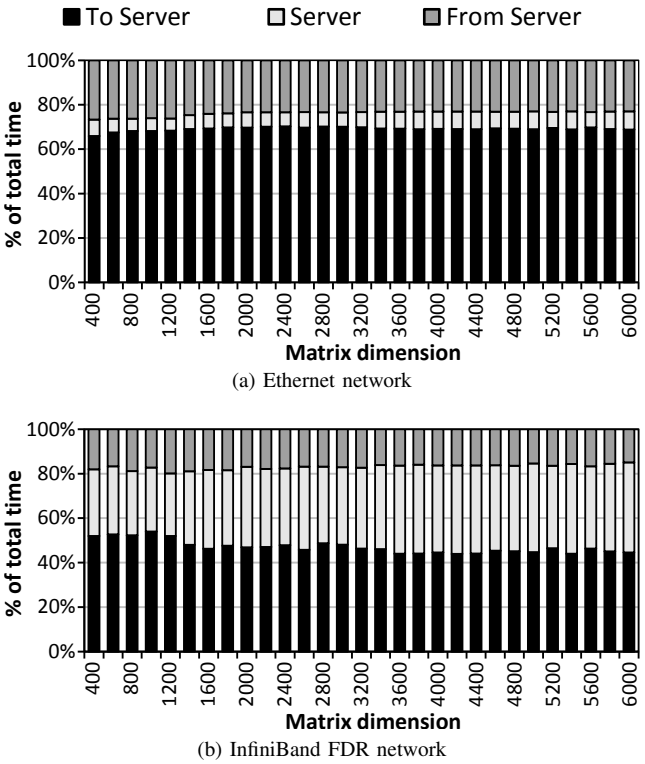


Figure 3: Performance of the DGEMM function.

function. Execution time when using the local CPU has been included as the reference. In order to perform a fair comparison, a powerful BLAS library should be selected for the local executions. In this regard, there are several good candidates, such as the GotoBLAS implementation, the OpenBLAS software, or the MKL library. Among these candidates we have selected the OpenBLAS library given its superior performance with respect to the other two incarnations of the BLAS library [18]. However, selecting a different candidate would not significantly modify the discussion in this section.

Execution times when using the local CPUs are referred by the curve labeled *"Xeon Local"*. Notice that the 12 available CPU cores in the system have been used in the local experiments. Figure 3 also depicts the execution time when using the new middleware along with an Ethernet network as well as the Infiniband QDR and FDR network fabrics (curves labeled *"Xeon ETH"*, *"Xeon QDR"*, and *"Xeon FDR"*, respectively). Results in Figure 3 clearly show that the performance of the network connecting the client and server nodes is crucial. In this regard, when 1 Gbps Ethernet is used, execution time is noticeably increased with respect to the execution time in the local CPUs. When the InfiniBand network is used, execution time for the new middleware is only slightly larger than the time required for the executions in the local CPU cores. Nevertheless, notice that times become similar for the largest matrix sizes. These results seem to point out that when the communication between client and server nodes is improved (next section), the new middleware may provide some performance gains.

There is an interesting detail in Figure 3 that is worth a comment. Notice that, surprisingly, both the QDR and FDR networks perform similar when using TCP/IP over InfiniBand. In order to verify this issue, we used the well known iperf tool [16] to gather bandwidth results. This tool showed that for the systems used in the experiments both interconnects provide basically the same performance, around 1190 MB/s.



(a) Ethernet network

(b) InfiniBand FDR network

Figure 4: DGEMM execution time breakdown.

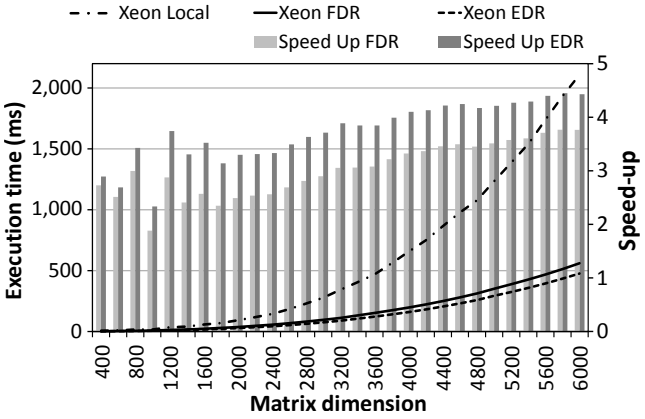
Figure 4 presents the breakdown of the execution time when using the new middleware. Time is broken down into three components: (1) time required to move the input data of the DGEMM function to the main memory of the remote server, (2) time spent in the server (which includes the actual computations as well as moving data from main memory to GPU memory and also moving results back from GPU memory to main memory), and (3) time required to return back the results to the client node. Labels *"To Server"*, *"Server"*, and *"From Server"* refer, respectively, to each of these components. It can be seen that when using the Ethernet network (Figure 4a), most of the total time is spent in moving data to/from the remote server, thus causing a noticeable increment in total execution time despite of using a powerful GPU to accelerate computations. The large time for data movement is diminished for the InfiniBand fabrics (Figure 4b), although communications still represent an important fraction of the total time, thus increasing total execution time with respect to local executions in the CPU cores, as shown in Figure 3.

Finally, in a similar way to the DGEMM function, where no performance gain was obtained due to the large communication overhead, in the case for the DGEMV and DDT functions, which present a smaller computation-to-communication ratio, no execution time reduction was achieved in our experiments.

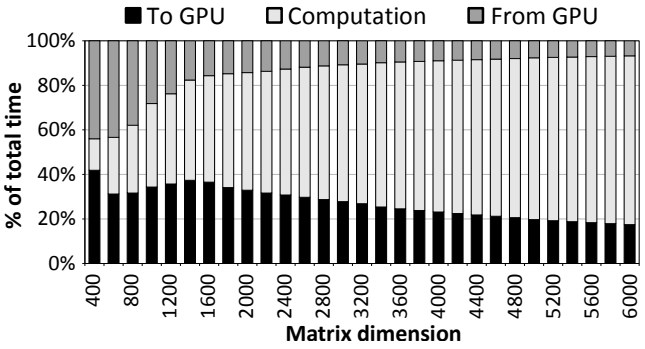
IV. Performance When Communications are Improved

In the previous section we have presented performance results of the new middleware, which makes use of the TCP/IP protocol stack to move data between client and server nodes. The communication layer currently available in the new middleware is very simple: data is moved from the client node to the server memory without any kind of optimization and, after receiving all the input data, they are moved to the GPU memory and then computation starts. However, it is possible to optimize such data movement in order to noticeably increase its performance. For example, the InfiniBand Verbs API may be used instead of the TCP/IP protocol stack, boosting network throughput. Also, an efficient communication pipeline could be leveraged, as in the rCUDA remote GPU virtualization framework [10]. Another possibility is using the GPU Direct RDMA mechanism provided by NVIDIA and Mellanox [17]. Moreover, notice that in these two options, data is directly moved from the main memory in the client node to the GPU memory in the remote node, thus not only making use of a higher network bandwidth, but also using an improved communication architecture. In this regard, it is important to remark that data in the previous section was sent from the client memory to the server memory and once all data arrived at the server memory, then it was moved to the GPU memory. Thus, data was moved in a stop&wait fashion, what introduced a large communication overhead. On the contrary, with the improved communication layer, data is directly moved from the client's main memory to the GPU memory at the server, avoiding the stop at the server's main memory. Additionally, data transmission is pipelined and therefore performance is not only improved because a higher network bandwidth is attained thanks to the use of the InfiniBand Verbs API, but also due to the fact that data transmission is not following a stop&wait approach but it is done in a cut-through way, thus also reducing communication latency.

Given that moving data from the memory in the client node to the GPU in the remote server is the main concern in the new framework, in this section we present an estimation of which would be the performance when an optimized communication larger is used. The performance estimation methodology consists in replacing, in the results presented in the previous section, the communication time between main memory in the client and the GPU memory in the server (including the intermediate stop at the server's main memory) by the time that an optimized communication layer would attain. Notice that for estimating the time required to move data to and from the remote server, which depends on the volume of input and output data and also on the network bandwidth attained for each transfer size, the bandwidth achieved by the rCUDA remote GPU virtualization framework [10] has been used instead of using the raw



(a) Execution time



(b) Time breakdown for InfiniBand EDR

Figure 5: DGEMM estimated performance.

bandwidth of the network fabric. This approach is more accurate than using the raw InfiniBand bandwidth because software layers always impose some loss to theoretical performance numbers. Additionally, we have considered in this section the use of InfiniBand FDR (56 Gbps) and also InfiniBand EDR (100 Gbps) network adapters.

Figure 5a presents the estimated performance for the DGEMM function. The label *"Xeon Local"* refers to the execution of the DGEMM function in the local CPU cores. The label *"Xeon FDR"* refers to the system configuration used in the previous section, whereas the label *"Xeon EDR"* refers to the same system configuration where the network adapter has been replaced by the Mellanox InfiniBand EDR card attaining 100 Gbps in total.

Figure 5a shows that the use of an optimized communication layer increments performance, almost achieving a 4x speed up when using InfiniBand FDR and 4.5x when InfiniBand EDR is used. Figure 5b presents the breakdown of the total estimated execution time when using the new middleware (only results for the EDR network adapter are shown due to space constraints). Notice that in this case the time breakdown is slightly different from the ones depicted in the previous section, where the time *"To Server"* denoted in previous sections the time required to move the input

data from the client node to the main memory of the remote server and the time *"Server"* included, in addition to the actual computation, the time to move the data from main memory in the server to the GPU memory. However, in Figure 5b, the time *"To GPU"* refers to the data movement from main memory in the client node directly to the GPU memory in the server whereas the time *"Computation"* only considers the actual execution of the DGEMM function in the GPU, excluding the data movement. It can be seen in Figure 5b that when an improved communication layer is used, most of the time is spent on the actual computations in the GPU instead of using most of the time in moving data.

Finally, remember that the DGEMM function analyzed above belongs to the Level 3 of BLAS, which comprises those functions with complexity $O(n^3)$. In this case, the new framework takes advantage of the high computation-to-communication ratio ($(n + 2)/3$ as discussed before). However, the DGEMV and DDOT functions present a much lower computation-to-communication ratio, what explains that in the estimations carried out with the improved communication layer (not shown), no performance gain is attained for these functions. Therefore, from now on we will focus on the DGEMM function.

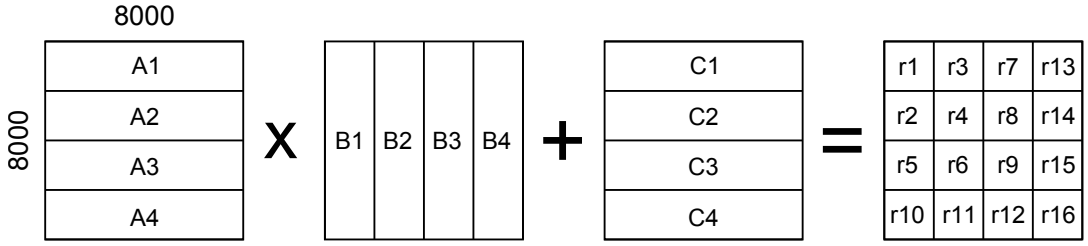
V. Performance when Computations are Pipelined

One may think about two possible ways of outsourcing the computations of the DGEMM function. In the first one, the naive approach, as soon as the wrapper for the DGEMM function is called by the application at the client node, the wrapper moves all the input data to the remote server and there the cuBLAS library is used to compute the DGEMM operation in the GPU. Once computations in the GPU are completed, results are returned back to the client side of the middleware, which forwards them to the original application. Although this naive implementation (used in the previous two sections) provides a 4x acceleration with respect to performing the DGEMM function in the local CPU cores, there is a smarter way to carry out the offloading process, which is based on overlapping data transmission with computations in a pipelined way. Figure 6 describes this process with a matrix example.

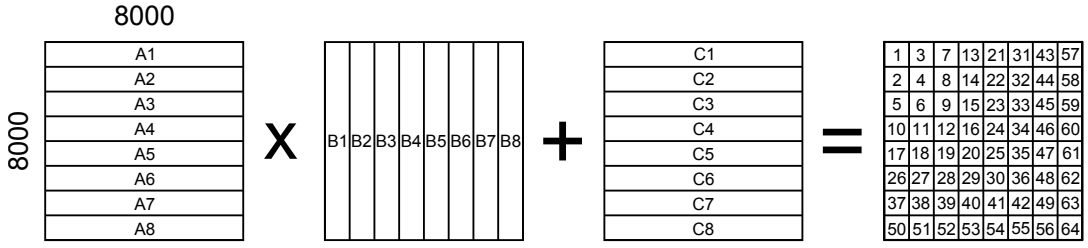
Figure 6a shows a DGEMM operation applied to 8000 by 8000 square matrices. It can be seen that matrices have been split into four blocks, each of them with dimensions equal to 2000x8000 elements. The key point about Figure 6a is that for computing a given element in the output matrix only some input data is required, but not the entire set of input data. For example, for computing those elements in the "r1" block of the output matrix, only blocks "A1", "B1", and "C1" are required. This allows appropriately organizing input data transmission so that computation in the server begins much before than the entire matrices have arrived, thus partially hiding data transmission time, which is the main concern in the proposed middleware. This is shown in the upper part of Figure 6c. In that figure it is shown that once the server has received blocks "A1", "C1", and "B1" then computation for the "r1" block can begin. In a similar way, once blocks "A2" and "C2" have arrived, results for block "r2" can be computed. Notice that transmission of blocks from the C matrix happens before than transmission of blocks from the B matrix. This reordering allows that computations for block "r2" start before receiving block "B2" from matrix B, thus saving some additional time. Another possible reordering could be first to forward block "B2" and then blocks "A2" and "C2". This would allow starting the computation of block "r3" even earlier without causing the gap shown in the figure after block "r1". However, this second ordering may cause a gap after computing "r3", given that blocks "A2" and "C2" are required before continuing with the computations. Finding an optimal transmission order is an open question that mainly depends on the relationship among transmission time and computation time, being both times dependent on block size.

Figure 6b shows a similar example when a smaller block size is leveraged. In this case, instead of splitting input matrices into four blocks, they have been divided into eight blocks. The bottom part of Figure 6c depicts the overlap between data transmission and computations in the GPU for this smaller block size. It can be seen that although transmitting each individual block requires less time, given their smaller size, the overall transmission time for the three matrices remains the same. However, using a smaller block size allows starting computations in the remote GPU earlier. This means that there is a bigger overlap between transmission of input data and computation. Hence, the main cause of overhead in the proposed middleware is reduced, making the proposal more appealing. Notice, however, that the use of many smaller blocks leads to an increased synchronization complexity and synchronization overhead, what may cancel part of the benefits of overlapping computations with transmission.

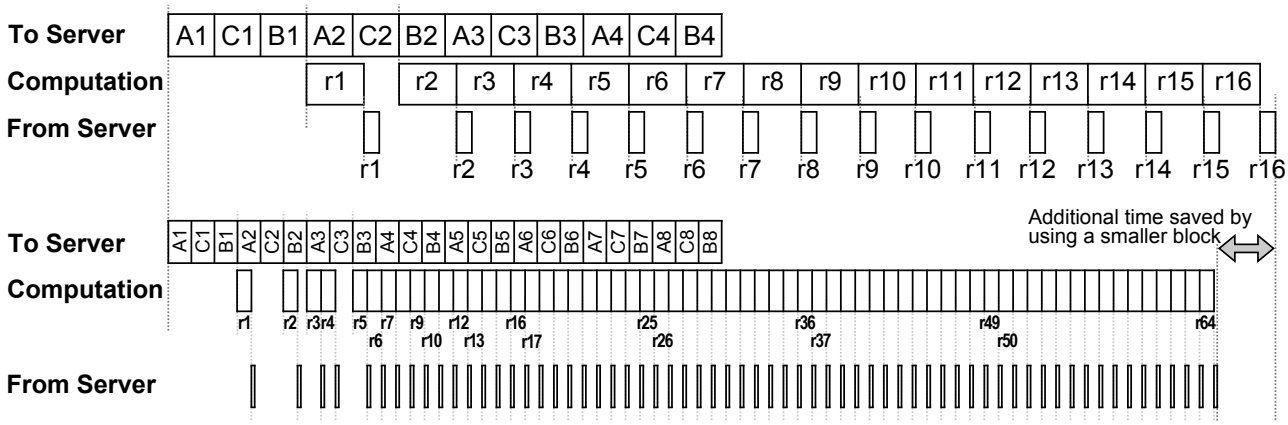
Figure 7 shows the estimated performance when the optimization discussed above is used. It can be seen that splitting the overall DGEMM computations into smaller operations reduces total execution time by 20% (sub-matrix dimension 8000 vs 1000). In summary, Figure 7 shows that by overlapping the transmission of data with computations the main concern of the proposed middleware, which is exchanging data with the remote server, is minimized. Notice, however, that computing the DGEMM function in the remote GPU takes longer than data transmission. Hence, a way to further accelerate the remote computation of the DGEMM function could be to make use of several GPUs and distribute the DGEMM computation among them by using the cuBLAS-XT library. This data and computation



(a) Diagram of DGEMM operation when matrices are split into four blocks.



(b) Diagram of DGEMM operation when matrices are split into eight blocks.



(c) Diagram of transmission and execution times for DGEMM when matrices are split into four and eight blocks.

Figure 6: Pipelining the DGEMM operation.

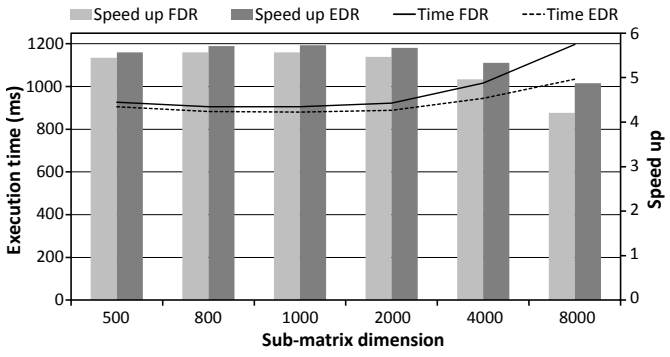
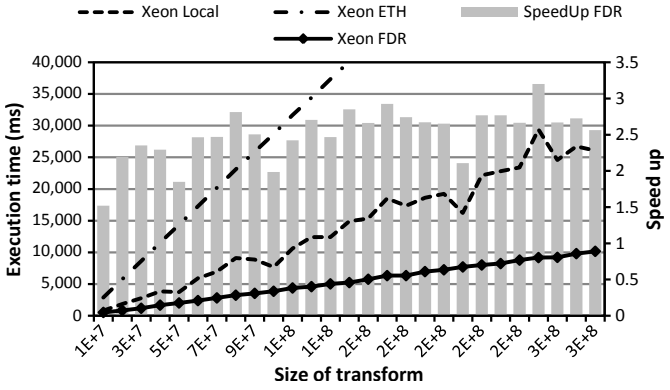


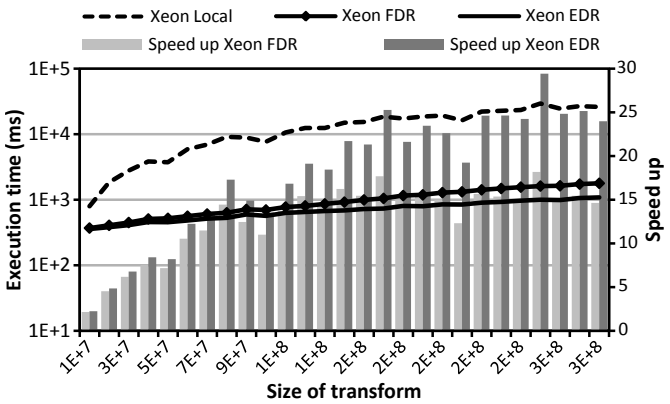
Figure 7: Performance estimation when the DGEMM operation in the remote GPU is pipelined. Square 8000x8000 matrices are used.

distribution is an open research issue still to be analyzed.

Finally, notice that every offloaded function presents a different behavior and therefore will achieve a different speed up. In this regard, the DGEMM function has attained up to 6x acceleration in the previous experiments. However, which should be the acceleration expected by other functions? Providing an answer is very difficult, given that it basically will depend on the exact function considered. As an example, Figure 8 shows the performance attained with the new middleware by a very different function: the discrete Fourier transform (DFT), belonging to the FFTW library. Figure 8a shows the performance obtained for different transform sizes when this function is offloaded with the new middleware using TCP/IP based communications, whereas Figure 8b presents a performance estimation when the communication layer is improved. It can be seen that



(a) Execution time with TCP/IP communications.



(b) Estimated performance with improved communications.

Figure 8: Performance of the DFT function from the FFTW library.

the DFT function behaves in a very different way to the DGEMM function. Actually, even with the TCP/IP based communications, this function presents 2.5x acceleration. Moreover, when the communication layer is enhanced, speed up grows up to 25x. Therefore, the conclusion is that for those functions that the computation-to-communication ratio is high enough, the new middleware may provide important accelerations.

VI. Conclusions

In this paper we have presented an early experience of a new middleware that outsources the CPU-based computations of mathematical libraries to remote accelerators. We have also conducted a performance evaluation by applying this middleware to some functions of the OpenBLAS and FFTW libraries. Results clearly show that this new framework is feasible and it may provide important reductions in execution time when the computation-to-communication ratio is large enough, being the bandwidth of the network fabric a key component to define the actual performance of this technique. Results have also shown that those functions, like DGEMV and DDOT, presenting a low computation-to-communication ratio should always be executed locally in the client node.

Future versions of this middleware, which will be made publicly available when completed, will completely support the OpenBLAS, FFTW and LAPACK libraries as well as the use of Intel Xeon Phi accelerators and an optimized communication layer.

References

[1] *VMware virtualization*, http://www.vmware.com

[2] *Xen Project*, http://www.xenproject.org

[3] *Kernel-based Virtual Machine*, http://www.linux-kvm.org

[4] *Oracle VM VirtualBox*, http://www.virtualbox.org

[5] A.A. Semnanian et al., *Virtualization Technology and its Impact on Computer Hardware Architecture*, ICIT 2011

[6] Mellanox, *ConnectX-3 VPI Single and Dual QSFP+ Port Adapter Card User Manual*, 2013

[7] Intel, *Intel Ethernet Server Adapter I350*, 2013

[8] *NVIDIA GRID Technology*, http://www.nvidia.com/object/grid-technology.html

[9] J. Song et al, *KVMGT: a Full GPU Virtualization Solution*, http://www.linux-kvm.org/wiki/images/f/f3/01x08b-KVMGT-a.pdf, 2014

[10] C. Reaño et al., *Boosting performance of GPU virtualization using InfiniBand Connect-IB and PCIe 3.0*, CLUSTER 2014

[11] G. Giunta et al., *A GPGPU Virtualization Component for High Performance Computing Clouds*, EuroPar 2010

[12] M. Oikawa et al., *DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment*, SCC 2012

[13] S. Iserte et al., *SLURM Support for Remote GPU Virtualization: Implementation and Performance Study*, SBACPAD 2014

[14] F. Silla, *rCUDA: Virtualizing GPUs to reduce cost and improve performance*, http://www.rcuda.net, 2014

[15] *BLAS (Basic Linear Algebra Subprograms)*, http://www.netlib.org/blas/

[16] *iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool*, https://github.com/esnet/iperf

[17] Mellanox, *Mellanox OFED GPUDirect RDMA Product Brief*, 2014

[18] Q. Wang et al, *Automatically generate high performance dense linear algebra kernels on x86 cpus*, in SC13

[19] NVIDIA, *cuBLAS library 7.0.* https://developer.nvidia.com/cuBLAS/, 2015.

[20] NVIDIA, *cuBLAS-XT library 7.0.* https://developer.nvidia.com/cuBLASXT/, 2015.

[21] E. Agullo et al, *Numerical linear algebra on emerging architectures*, Journal of Physics, 2009.

[22] T. A. C. Center, *GotoBLAS2* https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2/, 2014.

[23] FFTW, *FFTW (Fast Fourier Transform)*, http://www.fftw.org/, 2014.

[24] NVIDIA, *cuFFT library 7.0.* https://developer.nvidia.com/cuFFT/, 2015.

[25] Intel, *Intel Math Kernel Library (Intel MKL).* https://software.intel.com/en-us/intel-mkl/, 2014.