Performance of CUDA Virtualized Remote GPUs in High Performance Clusters

José Duato, Antonio J. Peña, Federico Silla
Universitat Politècnica de València (UPV)
46022 Valencia, Spain
{*jduato, fsilla*}*@disca.upv.es, apenya@gap.upv.es*

Rafael Mayo, Enrique S. Quintana-Ortí
Universidad Jaume I (UJI)
12071 Castellón, Spain
{*mayo, quintana*}*@icc.uji.es*

***Abstract*—In a previous work we presented the architecture of rCUDA, a middleware that enables CUDA remoting over a commodity network. That is, the middleware allows an application to use a CUDA-compatible Graphics Processor (GPU) installed in a remote computer as if it were installed in the computer where the application is being executed. This approach is based on the observation that GPUs in a cluster are not usually fully utilized, and it is intended to reduce the number of GPUs in the cluster, thus lowering the costs related with acquisition and maintenance while keeping performance close to that of the fully-equipped configuration.**

In this paper we model rCUDA over a series of high throughput networks in order to assess the influence of the performance of the underlying network on the performance of our virtualization technique. For this purpose, we analyze the traces of two different case studies over two different networks. Using this data, we calculate the expected performance for these same case studies over a series of high throughput networks, in order to characterize the expected behavior of our solution in high performance clusters.

The estimations are validated using real 1 Gbps Ethernet and 40 Gbps InfiniBand networks, showing an error rate in the order of 1% for executions involving data transfers above 40 MB. In summary, although our virtualization technique noticeably increases execution time when using a 1 Gbps Ethernet network, it performs almost as efficiently as a local GPU when higher performance interconnects are used. Therefore, the small overhead incurred by our proposal because of the remote use of GPUs is worth the savings that a cluster configuration with less GPUs than nodes reports.

***Keywords*-Clusters; CUDA; Graphics processors (GPUs); high performance computing; virtualization;**

I. Introduction

For many years, there have been attempts to exploit the massive parallel capabilities of Graphics Processors (GPUs) to accelerate specific parts of code using graphics-oriented languages such as OpenGL [1], [2] or Cg [3], [4]. Improvements in the programmability of these devices for general-purpose computing, known as GPGPU (General-Purpose computation on GPUs), have favored the fast adoption of these hardware accelerators in many application areas such as computational fluid dynamics [5] or computational algebra [6], to name only a few.

The way that programmers usually exploit GPUs is by off-loading parts of the application code that are computationally intensive to these devices. Although the programmer must specify which parts of the application are executed on the CPU and which parts are off-loaded to the GPU, the existence of libraries and frameworks as the ones mentioned above (CUDA) noticeably ease this task.

Due to the relatively high performance/cost ratio of GPUs, it is likely that in the near future large High Performance Computing (HPC) clusters will generalize the adoption of these devices, as many HPC applications employ GPUs as code accelerators. However, adding an accelerator to every node in an HPC cluster is not efficient neither from the performance point of view nor from the power consumption perspective —e.g., the power consumption of a GPU may well rate 25% of that of an HPC node. Additionally, in a configuration where all the nodes of a cluster are equipped with a GPU, it is unlikely that all GPU accelerators will be fully utilized all the time. Thus, reducing the number of accelerators in the cluster in order to increase their utilization could be interesting. Nevertheless, a configuration where not all the nodes in the cluster have an accelerator creates additional problems, as it requires a global scheduler to map tasks to nodes according to their hardware requirements. In summary, a configuration with less accelerators than nodes is less costly and more appealing, but it is not easy to manage unless other approaches are followed.

An alternative solution to deal with a cluster configuration equipped with less accelerators than nodes is virtualization. In the last years, hardware virtualization has become an important approach to reduce acquisition, administration, maintenance, space, and energy costs of HPC clusters and data processing centers. The idea behind this technique is to avoid attaching a given device to every node in a cluster by virtualizing those installed in a smaller number of nodes, which become servers that provide the services of the virtualized device to the rest of the cluster. Following this trend, virtualizing GPUs may solve the scheduling concerns of the cluster configuration mentioned earlier, as applications can now be dispatched to any node, independently of their hardware requirements. With this approach, it is possible to reduce related costs, hopefully reducing only slightly the performance of GPU-accelerated applications.

In order to make such distributed acceleration architecture possible, we have developed the rCUDA framework [7]. Our proposal, described in Section III, is based on the use of a

0190-3918/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPP.2011.58

middleware that includes a client and a server. The client is executed in all the machines of the cluster and provides the illusion of being a real GPU to applications requesting GPU services. Whenever one of those applications accesses the virtual GPU, the client forwards the request to the server owning the GPU. The server software, that is executed only in those cluster nodes having a GPU, forwards the request to the real GPU, sending back the corresponding results.

In our solution, the bandwidth between the main memory of the node requesting acceleration services and the memory of the remote GPU is limited by that of the network, since the bandwidth of the latter is, in general, lower than that of the PCI-Express (PCIe) bus used to connect the GPU and the network interface to their respective hosts. Fortunately, GPU codes are often compute-intensive and, therefore, I/O communication will not easily become a bottleneck.

The work presented in [7] focused on showing that remotely using CUDA devices is possible. That work introduced the rCUDA architecture without going into the performance details. In this paper we show that rCUDA is actually useful as its performance in HPC clusters may be close to local GPUs. To do so, our goal in this paper is to model the behavior of the CUDA Remoting Virtualization technique over some of the interconnects more frequently employed in HPC clusters, in order to analyze the feasibility of our proposed distributed accelerating architecture in high performance environments. For this purpose, we analyze the execution times of two different case studies in a commodity Gigabit Ethernet network and a 40 Gbps InfiniBand interconnect. From the analysis of these traces, we estimate the expected performance for these case studies in our target high performance networks; the estimations are validated employing the real interconnects. Finally, we compare the calculated performance with that of the executions in a regular local GPU using CUDA, and a local CPU using high performance libraries, in order to determine the cost of using a remote GPU to accelerate the executions. The overall result is an in-depth analysis of how our rCUDA proposal is influenced by the performance of the underlying network available in the cluster. Indirectly, we show the feasibility of virtualized GPUs in clusters.

The rest of the paper is organized as follows. After reviewing related work in Section II, our virtualized GPU architecture is briefly introduced in Section III. The experimental framework is detailed in Section IV, while Section V defines the model used in Section VI to estimate the performance of our solution in a variety of high performance networks. Section VII summarizes the conclusions of this work.

II. Related Work

There are several upcoming commercial solutions that allow multiple servers in a rack to share a few GPUs —such as NextIO's N2800-ICA [8] by PCIe virtualization [9]— enabling a cluster configuration where only a few of its nodes have a GPU. However, they do not allow multiple nodes concurrently accessing the same GPU, as GPUs need to be assigned to a specific node at any time. Moreover, these proprietary hardware-supported solutions are non-standard and often substantially expensive.

On the other hand, many software efforts to virtualize GPUs have been carried out. In this case, due to the fact that the low level protocols used to address GPUs are proprietary and strictly closed by the GPU vendors, the virtualization boundary has to be placed on top of the open high-level Application Programming Interfaces (APIs), such as Microsoft's Direct3D [10] and OpenGL [2], for graphics acceleration; or CUDA [11] and OpenCL [12] for GPGPU.

In the case of virtualizing GPUs for graphics accelerators, there are several works in the field of virtual machines (VMs) such as, e.g., VMware's Virtual GPU [13], a specific GPU virtualization architecture developed for VMware's hosted products; and VMGL [14], an OpenGL virtualization solution both virtual machine monitor and GPU independent.

When GPUs are used as accelerators for GPGPU, there is no need to take care of visual output nor virtual machine-related issues, and the specifics of virtualizing GPUs change drastically. This way of using GPUs has led to different solutions virtualizing the CUDA Runtime API in VM environments, resulting in the vCUDA [15] and GViM [16] prototypes; and gVirtuS [17], targeting the field of cloud computing. VCL [18] is a similar tool for the OpenCL API.

The software we use for this study, rCUDA, is a solution specifically designed to run in an HPC cluster environment. Given the similarities of the different GPGPU remoting tools, we expect no major differences in their network-dependent performance behavior, as long as the application-level communication protocol is well designed and its implementation properly tuned. However, the particular target environment of each software package prevent us to perform a comparison in this study, which targets the HPC cluster environment using the CUDA Runtime API.

The main contributions of this work are (1) providing a performance model for simple applications employing virtualized remote GPUs through HPC interconnects, and (2) demonstrating that current network technologies offer a performance that allows running HPC applications following our proposal. Note that only applications making use of synchronous data transfers are covered by the developed estimation model, leaving asynchronous transfers for future work. Similarly, potential network contention caused by multiple applications running in a cluster featuring several GPGPU servers will also be covered in future work.

III. Background on rCUDA

Following our proposed distributed acceleration architecture for HPC clusters with GPUs attached only to a few of its nodes (see Figure 1), when a node without a local GPU executes an application that makes use of a GPU to

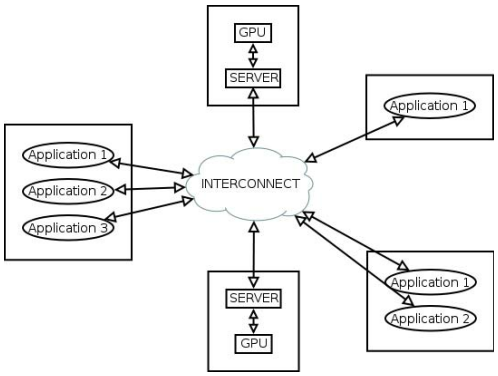


Figure 1. Distributed acceleration architecture.

accelerate part of its code (usually referred to as *kernel*), some support has to be provided to deal with the data and code transfers between the local main memory and the remote GPU memory, as well as the remote *kernel* execution.

In a previous work [7], we presented a middleware solution to support such a virtualized GPU architecture targeting the NVIDIA CUDA programming environment. In that paper, we reported some preliminary experimental results based on two simple case studies running on two nodes connected through a Gigabit Ethernet interconnect.

That framework, named rCUDA, is designed following the client-server distributed architecture: on one side, clients employ a library of wrappers to the CUDA Runtime API and, on the other side, there is a GPU network service listening for requests on a TCP port. Figure 1 illustrates this proposal, where several nodes running different GPU-accelerated applications can concurrently make use of the whole set of accelerators installed in the cluster. When an application demands a GPU service, its request is delivered to the client side of our architecture, running in that computer. The client forwards the request to one of the servers, which accesses the GPU installed in the server computer and executes the request in it. Time-multiplexing (sharing) the GPU is accomplished by spawning a different server process for each remote execution over a new GPU context.

In our solution, the client side sends a message to the server for each CUDA call performed by the application requesting GPU services. In these messages, the first 32 bits of the request identify the specific CUDA function called, while the subsequent data is function-dependent and specifies the particular parameters of each function call. The server always sends a 32-bit result code of the operation, and possibly more data depending on each particular function.

Table I shows the fields and their corresponding sizes (in bytes) that are transferred for the most commonly used operations. The x on this table represents the data size in the cases where it depends on the specific operation.

In general, the execution of a kernel in our approach requires the following phases, illustrated in Figure 2 using

Table I
BREAKDOWN OF SOME REMOTE API MESSAGES

Operation	Field	Send (bytes)	Receive (bytes)
Initialization	Compute capability		8
	Size	4	
	Module	x	
	CUDA error		4
	Total	$x+4$	12
cudaMalloc	Function id.	4	
	Size	4	
	CUDA error		4
	Device pointer		4
	Total	8	8
cudaMemcpy	Function id.	4	
	Destination	4	
	Source	4	
	Size	4	
	Kind	4	
(to device)	Data	x	
	CUDA error		4
(to host)	Data		x
(to device)	**Total**	$x+20$	4
(to host)	**Total**	20	$x+4$
cudaLaunch	Function id.	4	
	Texture offset	4	
	Parameters offset	4	
	Number of textures	4	
	Block dimension	12	
	Grid dimension	8	
	Shared size	4	
	Stream	4	
	Kernel name	x	
	CUDA error		4
	Total	$x+44$	4
cudaFree	Function id.	4	
	Device pointer	4	
	CUDA error		4
	Total	8	4

the matrix-matrix product, $C = A \cdot B$, as an example:

1) **Initialization stage.** The client side automatically establishes a connection with the remote server, and locates and sends the *GPU module* of the application requesting GPU services, which comprises the code to be executed on the GPU (kernels) and other related information such as statically allocated variables.
2) **Memory allocation.** The client requests memory allocation on the GPU for the data accessed by the kernel.
3) **Input data transfer.** If there is additional data to be used, it must be transferred from client to server.
4) **Kernel execution.** Remote GPU code execution.
5) **Output data transfer.** Once the kernel execution is completed, the output data is available to be sent back.
6) **Memory release.** GPU-allocated memory is released.
7) **Finalization stage.** The client application closes the socket. The daemon server quits servicing the current execution and releases the associated resources.

Our middleware provides applications with the illusion that they are dealing with a real GPU. This solution also enables concurrent access to the CUDA-compatible devices in a way that is completely transparent to the programmer.

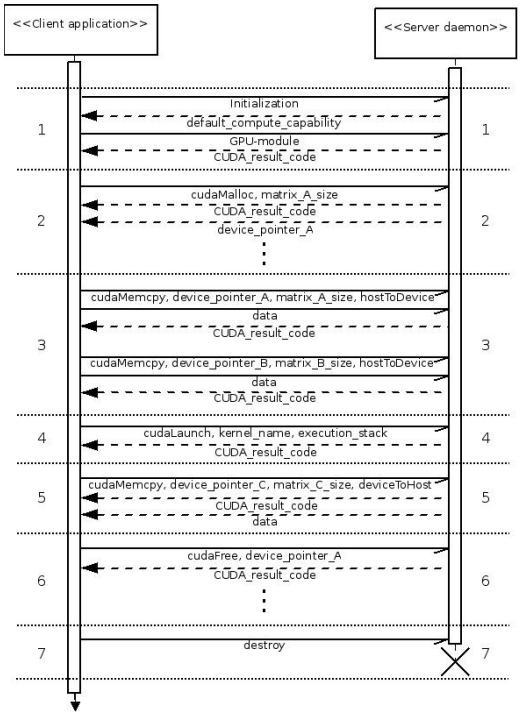


Figure 2. Client-server communications for a matrix multiplication.

IV. Experimental Framework

As mentioned above, the goal of this work is to assess how the network interconnecting the nodes of the cluster affects the performance of our proposal. More specifically, as HPC clusters usually employ high performance networks, our goal is to analyze how the characteristics of these networks influence the behavior of our distributed acceleration architecture. In this section we discuss the experimental environment for our tests by presenting first the setup of the system, and then introducing the case studies that are evaluated in the experimental results. Those results will later be used in Section V to create a model in order to finally assess the performance of the target HPC networks in Section VI.

A. System

We have conducted our experiments using two nodes, each equipped with two Quad-Core Intel® Xeon® E5520 processors featuring 2.27 GHz and 24 GB of main memory. The nodes run Linux OS (kernel 2.6.18). The GPU is an NVIDIA Tesla C1060 (driver version 190.18 Beta) attached to a PCIe 2.0 x16 port in one of the nodes. The server daemon is built over CUDA Toolkit 2.3. Note that by using just two nodes, several concerns, like network congestion, or congestion in the rCUDA server when being concurrently accessed by several clients, cannot be analyzed. Nevertheless, the focus of this work is analyzing how the smaller bandwidth of a cluster interconnect, with respect to the on-board PCIe bus, affects the performance of rCUDA.

The interconnects in our system are a 1 Gbps Ethernet (GigaE) and a 40 Gbps InfiniBand (40GI). In order to characterize the interconnect latencies in our system, we carried out several tests. In Figure 3 we show the end-to-end latency of the GigaE network for a range of data payload sizes, measured with a customized ping-pong test via standard TCP sockets. To reproduce the way our middleware handles the data transmissions, we disabled the TCP-layer congestion control algorithm. In particular, we explicitly control the instant a frame must be sent out by modifying a series of TCP options and policies, in order to avoid unnecessary delays introduced by the default congestion control algorithm in this protocol (Nagle's algorithm [19]).

The left-hand plot in Figure 3 shows the latency for small packets. To deal with network variability, the results correspond to the average of 250 executions (a maximum standard deviation of 22.7 μs was observed). We can observe a non-linear time response with the data payload, as for small data transfers, the TCP window size and, therefore, the number of TCP frames and ACKs that have to be transmitted, introduce a delay that cannot be hidden.

The right-hand plot in the same figure reports the end-to-end latency for large data payloads. The results in this plot are the minimum of 100 executions (maximum standard deviation of 2.1 ms). These results show that we can assume a linear end-to-end latency proportional to the payload size, provided that this size is large enough. Hence, performing a linear regression of the data, we obtain a function f to approximate the transfer time (in ms) for n MB of data payload, featuring a correlation coefficient of 1.0, as follows:

$$f(n) = 8.9n - 0.3$$

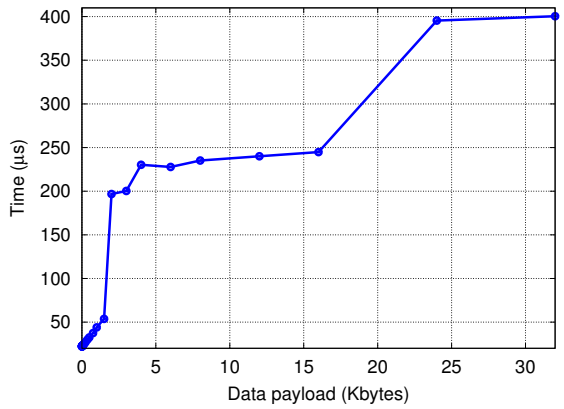
Note that we can approach a linear end-to-end latency despite the per-packet timing irregularities shown in the left-hand plot for small data payload sizes because these can be hidden when a larger amount of TCP frames are involved in the transmission, as the transmission window increases to its maximum size, thus stabilizing the results.

Figure 4 shows the corresponding results for the 40GI network. The obtained standard deviations are 1.1 μs for small data payloads and 4.8 ms in the case of large data transfers. In this case, the left-hand plot shows a more linear response in comparison with the GigaE network, due to the underlying InfiniBand protocol. The linear regression function g of the end-to-end latency (in ms) for large data transfers, to approximate the transfer time in ms for n MB of data payload, in this case is:

$$g(n) = 0.7n + 2.8$$

featuring again a correlation coefficient of 1.0.

These simple ping-pong tests reveal a maximum effective one-way throughput of 112.4 MB/s over the GigaE network, while the resulting rate for the 40GI is 1,367.1 MB/s. Our tests also reveal a peak effective bandwidth for the data



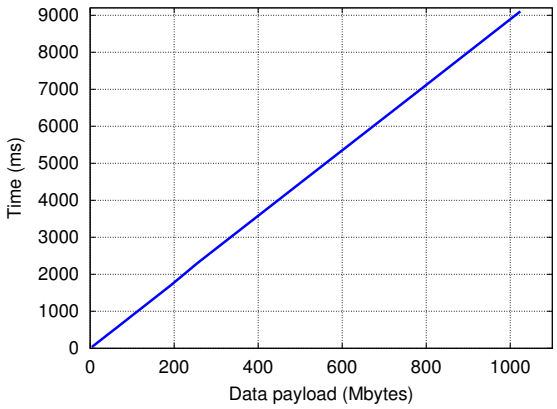
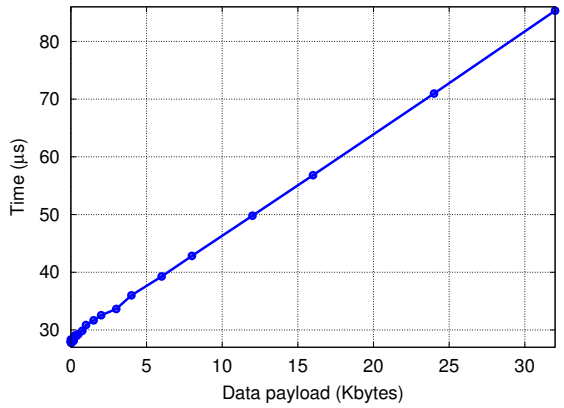


Figure 3. End-to-end latency on the GigaE network. **Left:** small packets. **Right:** large data payloads.



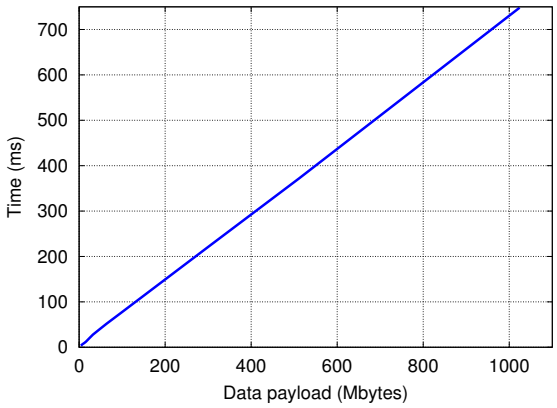


Figure 4. End-to-end latency on the 40GI network. **Left:** small packets. **Right:** large data payloads.

transfers between the memory of the local GPU and the main memory of the computer of 5,743 MB/s —a PCIe 2.0 x16 graphics link offers a maximum bandwidth of 8 GB/s. Therefore, it is obvious that the bottleneck for the data transfers is located in the network interconnect.

B. Case Studies

The impact of the virtualization overhead caused by the interconnect is evaluated in two case studies: the matrix product (MM) and the 1D Fast Fourier Transform (FFT).

Intel MKL (v10.1) and FFTW (v3.2.2) are employed on the CPU for the matrix-matrix product and the FFT, respectively. On the GPU, we employ Volkov's implementations of the matrix-matrix product routine [20] and FFT. The sizes of the GPU modules transferred to the remote GPU to execute the case studies are 21,486 and 7,852 bytes for the matrix-matrix product and FFT, respectively. To exploit the GPU massive parallel capabilities, our tests comprise different numbers of parallel FFT operations.

Table II shows the message sizes and the estimated transfer times for the remote CUDA Runtime API calls involved in our case studies, given the latencies shown in Figures 3 and 4, and the message sizes from Table I. In this table, m represents the dimensions of the matrices involved in the matrix-matrix product (all matrices are square) and n is the number of FFTs to compute in parallel (also known as `batch`). Note that one data element in the matrix-matrix product occupies 4 bytes (single precision real floating-point data), while in the FFT it requires 8 bytes (single precision floating-point complex points). Thus, the number of data bytes for each memory transfer operation in the first case study is $4m^2$. For the second case study, it is $(8 \times 512)n$, as we compute 512 points on each FFT operation. The transfer times of the operations involving small fixed-size data transfers have been directly extracted from the real measured times represented in the left-hand side plots in Figures 3 and 4 (interpolated if the exact value was not available). On the other hand, the costs of the memory transfer operations are extracted from the presented end-to-end latency equations ($f(n)$ and $g(n)$), as they are expected to involve sufficient data to assume a linear latency, as that shown in the right-hand side plots in Figures 3 and 4.

Table II
ESTIMATED TRANSFER TIMES FOR THE REMOTE API CALLS INVOLVED IN THE CASE STUDIES

Case study	Operation	Data size (bytes)		Transfer time (μs)			
				GigaE		40GI	
		Send	Receive	Send	Receive	Send	Receive
MM	Initialization	21490	12	338.7	44.4	80.9	20.0
	cudaMalloc ($\times 3$)	8	8	22.2	22.2	27.9	27.9
	cudaMemcpy ($\times 2$)	$4m^2 + 20$	4	$35.6m^2 + 177.7$	22.2	$2.8m^2 + 16.8$	27.9
	cudaLaunch	52	4	23.1	22.2	27.9	27.9
	cudaMemcpy	20	$4m^2 + 4$	22.4	$35.6m^2 + 35.3$	27.8	$2.8m^2 + 5.6$
	cudaFree ($\times 3$)	8	4	22.2	22.2	27.9	27.9
	Total	$8m^2 + 21650$	$4m^2 + 64$	$71.2m^2 + 872.8$	$35.6m^2 + 279.5$	$5.6m^2 + 337.6$	$2.8m^2 + 276.7$
FFT	Initialization	7856	12	233.9	44.4	39.5	20.0
	cudaMalloc	8	8	22.2	22.2	27.9	27.9
	cudaMemcpy	$4096n + 20$	4	$36454.4n + 177.7$	22.2	$2867.2n + 16.8$	27.9
	cudaLaunch	58	4	23.2	22.2	27.9	27.9
	cudaMemcpy	20	$4096n + 4$	22.4	$36454.4n + 35.3$	27.8	$2867.2n + 5.6$
	cudaFree	8	4	22.2	22.2	27.9	27.9
	Total	$4096n + 7970$	$4096n + 36$	$36454.4n + 501.6$	$36454.4n + 168.5$	$2867.2n + 167.8$	$2867.2n + 137.2$

V. ESTIMATION MODEL

The results shown in Table II reveal that the transfer times for all of the operation calls are negligible, except for the memory transfer operations (`cudaMemcpy`) because of the relatively low bandwidth of the communication channel in comparison with that of the PCIe bus. For our estimation model, we will also assume that the time required to open/close a socket is negligible, provided that the size of the problem is large enough. Therefore, in the following we use the data payload transfer time of the memory transfer operations as an approximation for the overhead introduced by the network in our distributed computing system with virtualized remote GPUs. This approximation is reinforced by the fact that network latency has little impact on the timing results of our case studies due to the small number of messages transferred across the network. Thus, we will neglect times involving small data payloads and will approximate the overhead focusing on memory transfer operations.

Table III details the estimated data transmission times for each memory copy operation for a range of problem sizes of our case studies, given the effective one-way bandwidth of both networks. `Dim` (or `batch`) refer to the problem size, while `data` refers to the amount of data (in MB) transferred in each operation. Numbers in this table present the estimated times for single memory transfer operations. Thus, in order to know how much time each case study spends on all the data transmissions, these transmission times must be multiplied by 3 in the case of the matrix-matrix product, as this execution comprises 2 memory transfers for the input data and an additional transfer to retrieve the output data; and must be multiplied by 2 in the case of the FFT, as there is one memory transfer per direction.

In order to extract an estimation model for the execution times of our case studies when executed over different networks, we subtract the total estimated transfer times (presented in Table III, and appropriately multiplied by 2 or by 3) from the real execution times using the GigaE and 40GI networks shown in Table IV. Thus, we obtain a *fixed* time to be added to the estimated transfer times of each target network. This fixed time is spent on the computations over the CPU and GPU —including those corresponding to the management tasks needed by our middleware, random data generation, rCUDA initialization, etc.— on data transfers across the PCIe, and it also includes other negligible transfer times (see Section IV). Therefore, we assume that porting our solution to different networks will not significantly change the time spent by our middleware in the management of the communications and hence this fixed time will not significantly differ. This can be seen in Table IV for the MM case study, where it is shown that for a problem comprising large data transfers, this fixed time is almost independent of the actual interconnect. However, the table shows how for the FFT, a problem with smaller data transfers, this fixed time across different interconnects presents larger variability.

With this method, we obtain two estimation models derived from each initial network. Both models are cross-validated in Table IV, i.e., the execution timings estimated with the model obtained from the GigaE network are compared with the real execution times using the 40GI interconnect, and vice versa. This table details the execution

Table III
ESTIMATED TRANSFER TIMES (IN MS) FOR EACH MEMORY COPY ON OUR NETWORKS — DATA SIZE IN MB

MM				FFT			
Dim.	Data	GigaE	40GI	Batch	Data	GigaE	40GI
4096	64	569.4	46.8	2048	8	71.2	5.9
6144	144	1281.1	105.3	4096	16	142.3	11.7
8192	256	2277.6	187.3	6144	24	213.5	17.6
10240	400	3558.7	292.6	8192	32	284.7	23.4
12288	576	5124.6	421.3	10240	40	355.9	29.3
14336	784	6975.1	573.5	12288	48	427.0	35.1
16384	1024	9110.3	749.0	16384	64	569.4	46.8
18432	1296	11530.2	948.0				

Table IV
CROSS-VALIDATION OF BOTH ESTIMATION MODELS FOR THE CASE STUDIES — TIMES IN SECS (MM) AND MS (FFT)

Case study	Size	GigaE model				40GI model			
		Measured		Estimated	Error	Measured		Estimated	Error
		GigaE	Fixed	40GI		40GI	Fixed	GigaE	
MM	4096	3.64	1.93	2.08	2.16%	2.03	1.89	3.60	-1.21%
	6144	8.47	4.62	4.94	1.76%	4.85	4.54	8.38	-1.01%
	8192	15.60	8.77	9.33	-0.10%	9.34	8.78	15.61	0.06%
	10240	25.47	14.79	15.67	-0.41%	15.74	14.86	25.54	0.25%
	12288	38.39	23.02	24.28	-0.54%	24.42	23.15	38.53	0.35%
	14336	54.96	34.03	35.75	0.73%	35.49	33.77	54.70	-0.47%
	16384	74.13	46.80	49.04	-1.78%	49.93	47.68	75.02	1.20%
	18432	97.65	63.06	65.90	-1.72%	67.05	64.21	98.80	1.18%
FFT	2048	354.33	211.98	223.69	33.95%	167.00	155.30	297.65	-16.00%
	4096	555.67	270.97	294.38	30.26%	226.00	202.59	487.29	-12.31%
	6144	761.00	333.95	369.06	20.48%	306.33	271.22	698.27	-8.24%
	8192	964.33	394.94	441.75	16.35%	379.67	332.85	902.25	-6.44%
	10240	1167.67	455.92	514.44	12.32%	458.00	399.48	1111.23	-4.83%
	12288	1371.33	517.24	587.46	9.26%	537.67	467.45	1321.54	-3.63%
	16384	1782.00	643.21	736.84	5.77%	696.67	603.04	1741.83	-2.25%

times of our case studies over the source network for each model, as well as the calculated fixed times for every problem size. The differences in the fixed times for both models are mostly attributed to unexpected network transfer times related to the TCP window status. The table also shows the expected execution times of the case studies for the validation network, as well as the resulting error rates between the estimations and the real executions. The empirically measured times are averaged from 30 executions (a maximum standard deviation of 1.0 s was observed in the case of the matrix-matrix product and 14.4 ms for the FFT).

The absolute error rates of the estimations reported in these tables are below 2.2% in our first case study, where the data amount transferred in each execution is between 192 MB and 3.8 GB. In the case of the FFT, we experienced higher absolute error rates —between 2.5% and 33.9%— as the amount of transferred data per execution (16–128 MB) is lower than in the prior case, and the TCP-related distortions become more relevant; see Section IV. Thus, the accuracy of our method has been validated for large datasets. As both GigaE- and 40GI-based models present relatively low error rates for large datasets, both of them are considered in Section VI to estimate the performance of our case studies when using a remote GPU accelerator over different HPC networks. In the case for small datasets, the error is considerably large. However, as it will be shown in next section, applications presenting small datasets may not even be eligible for being accelerated in a local GPU. Therefore, that case presents much less interest. It is included in the paper for the sake of completeness.

VI. MODELING HPC NETWORKS

Using the analysis from the previous section, we estimate the performance of our case studies over a series of HPC interconnects. This study considers the following networks: (1) 10-Gigabit iWARP Ethernet (10GE), (2) 10 Gbps InfiniBand (10GI), (3) Myrinet-10G (Myr), and (4) HyperTransport networks based on FPGA (F-HT) and ASIC (A-HT).

The goal of the experiments in this section is to assess the performance of the case studies when executed in an environment using our solution over these networks, and to compare it with the execution time required by a system with only a local GPU, and also with the execution on a local CPU employing high performance libraries.

For the 10GE, 10GI, and Myr networks, we consider the configurations and performance (at the user level) reported by Rashti and Afsahi [21], where the bandwidth is extracted from the measured round-trip time divided by two.

A. The Networks

The target 10GE network is based on NetEffect NE010e 10-Gigabit Ethernet channel adapters and features a one-way effective bandwidth of 880 MB/s.

The 10GI network employs dual-port 10Gb/s Mellanox MHEA28-XT HCA cards. The one-way effective bandwidth for this network is reported to be “roughly 970 MB/s” —in the following we use this value.

For the Myr network, Myri-10G NICs (10G-PCIE-8A-C) were used. We will consider an effective bandwidth of 750 MB/s for this interconnect.

The HyperTransport-based networks are part of an ongoing project based on the new extensions to the HyperTransport technology, recently proposed in the High Node Count HyperTransport Specification [22]. This specification provides a non-coherent shared memory map for a large number of computing nodes with very low latency, provided that data transfers are managed by hardware without the intervention of the OS kernel. For the F-HT, we consider a 16-bit link at 400 MHz, resulting in a total bandwidth of 12.8 Gb/s. In order to calculate the effective bandwidth, we have assumed the maximum packet size: 64 bytes with 8 bytes of header (therefore efficiency is 88%). For the A-HT

Table V
ESTIMATED TRANSFER TIMES (IN MS) FOR EACH MEMORY COPY ON THE TARGET NETWORKS — DATA IN MB

Case	Size	Data	10GE	10GI	Myr	F-HT	A-HT
MM	4096	64	72.7	66.0	85.3	44.4	22.2
	6144	144	163.6	148.5	192.0	99.9	49.9
	8192	256	290.9	263.9	341.3	177.5	88.8
	10240	400	454.5	412.4	533.3	277.4	138.7
	12288	576	654.5	593.8	768.0	399.4	199.7
	14336	784	890.9	808.2	1045.3	543.7	271.8
	16384	1024	1163.6	1055.7	1365.3	710.1	355.1
	18432	1296	1472.7	1336.1	1728.0	898.8	449.4
FFT	2048	8	9.1	8.2	10.7	5.5	2.8
	4096	16	18.2	16.5	21.3	11.1	5.5
	6144	24	27.3	24.7	32.0	16.6	8.3
	8192	32	36.4	33.0	42.7	22.2	11.1
	10240	40	45.5	41.2	53.3	27.7	13.9
	12288	48	54.5	49.5	64.0	33.3	16.6
	16384	64	72.7	66.0	85.3	44.4	22.2

we assume that we will be able to double the bandwidth, although preliminary studies report an acceleration factor of up to 5x. Thus, the one-way effective bandwidths expected for these networks are 1,442 MB/s for the solution based on FPGA and 2,884 MB/s for the A-HT.

B. Estimated Performance

Similarly to Table III for our real interconnects, Table V details the estimated data transmission times for each memory copy operation for a range of problem sizes of our case studies on our target HPC networks, given their effective one-way bandwidth. As can be derived from these results, the theoretical transmission time reduction when using high performance networks yields up to a 96% for the A-HT interconnect in comparison with the GigaE network.

In addition to estimate the transmission times, we can also estimate the execution times of our case studies when executed over the HPC target networks employing the models created in Section V. Table VI shows the expected execution times of the case studies for each one of the target networks using both the GigaE- and 40GI-based estimation models. Additionally, the measured times of the executions on the local CPUs and GPUs are also detailed, as well as those of the executions using remote GPUs over the real GigaE and 40GI interconnects. The local CPU executions are carried out using the 8 cores in both case studies. As expected from the results presented in Section V, the estimations based on both models present small differences for large datasets, i.e. MM, while the estimations for FFT, with much smaller data transfers, present more dispersed values. Note that for the MM data, in the case of a matrix of 4096×4096 elements, the execution on the local GPU is slower than that on a remote accelerator over the 40GI network and the estimations for our target networks. The reason is that the rCUDA daemon pre-initializes the CUDA context, thus avoiding the CUDA environment initialization delay.

Finally, in Figures 5 and 6 we show the execution times presented in Table VI, as a graphical representation of the behavior of our proposal on different networks. As can be observed in the figures, there are no major differences between the estimations based on both models.

The left-hand plots in Figures 5 and 6 also report that computing a matrix-matrix product over a virtualized remote GPU employing high performance networks is only slightly slower than doing it directly on a local GPU, provided that the problem is large enough —the asynthotic computational cost of this problem is $O(m^3)$, where m is the problem size. Additionally, results clearly show that in this case using a remote GPU instead of a high performance 8-core processor is worth, even despite the communications overhead. However, in the case of the FFT (right-hand plots in Figures 5 and 6), the overhead is more considerable, as the problem is not so computationally intensive —the FFT computational cost is $O(n \log n)$, where n is the problem size— and therefore it is not well suited for its execution in a remote GPU. Actually, note that the FFT problem is not only not well suited for GPU remoting, but it is also not eligible for being accelerated in a local GPU, as can be derived from the execution times in a local CPU being lower than in a local GPU. The reason is that for the FFT algorithm, despite the performance speed-up obtained by the execution of the FFT GPU-kernel in comparison with the CPU execution, the data transfer time, even across the PCIe bus, adds an overhead that makes the GPU execution for the FFT slower than that on the CPU if the data is not previously available on the GPU memory (i.e., if the FFT is not part of a more complex algorithm).

From these results we can predict whether a case study will benefit from using a remote accelerator on the target networks instead of the local CPUs. As presented in Figures 5 and 6 for the GigaE- and 40GI-based models, respectively, our estimations reveal that it is worth using a remote accelerator over all the studied HPC networks in the case of the matrix-matrix product. In the case of the FFT application, this case study illustrates the overhead of using remote GPUs on solving algorithms featuring low computational costs. In general, our study reveals that if the problem is well suited to be accelerated in a local GPU, then the overhead of using a remote GPU will be worth the cost reduction provided by such a configuration.

VII. CONCLUSION

Incorporating one GPU-based code accelerator in every node of an HPC cluster is costly. Instead, we advocate for a hybrid configuration where only a few nodes of the cluster are equipped with accelerators, and these devices are shared by all the cluster nodes.

In this paper we have analyzed the performance of two different case studies over real networks, and from these results we have estimated their performance when running in our virtualized GPU distributed system equipped with a

Table VI
MEASURED VS. ESTIMATED EXECUTION TIMES OVER SEVERAL NETWORKS (MM IN S; FFT IN MS)

Case study	Size	Measured				Estimated									
						GigaE model					40GI model				
		CPU	GPU	GigaE	40GI	10GE	10GI	Myr	F-HT	A-HT	10GE	10GI	Myr	F-HT	A-HT
MM	4096	2.08	2.40	3.64	1.93	2.13	2.15	2.19	2.07	2.00	2.09	2.11	2.15	2.02	1.96
	6144	5.66	4.58	8.47	4.62	5.07	5.11	5.20	4.92	4.77	4.98	5.03	5.11	4.84	4.69
	8192	11.99	8.12	15.60	8.77	9.56	9.64	9.79	9.30	9.04	9.57	9.65	9.80	9.31	9.05
	10240	21.52	13.30	25.47	14.79	16.03	16.16	16.39	15.63	15.21	16.10	16.22	16.46	15.69	15.27
	12288	35.45	20.37	38.39	23.02	24.80	24.98	25.32	24.22	23.62	24.93	25.12	25.46	24.35	23.75
	14336	54.00	29.64	54.96	34.03	36.46	36.70	37.17	35.66	34.85	36.20	36.44	36.91	35.40	34.59
	16384	78.87	41.43	74.13	46.80	49.96	50.29	50.89	48.93	47.86	50.85	51.18	51.78	49.81	48.75
	18432	109.12	55.86	97.65	63.06	67.06	67.47	68.24	65.75	64.40	68.22	68.63	69.39	66.90	65.56
FFT	2048	41.67	51.00	354.33	167.00	228.48	230.17	233.32	223.08	217.53	171.79	173.48	176.63	166.39	160.84
	4096	74.67	102.33	555.67	226.00	303.96	307.33	313.64	293.16	282.06	235.58	238.96	245.26	224.78	213.69
	6144	115.67	153.33	761.00	306.33	383.44	388.50	397.95	367.24	350.60	320.71	325.77	335.22	304.51	287.87
	8192	150.33	201.67	964.33	379.67	460.92	467.67	480.27	439.32	417.13	398.83	405.58	418.19	377.24	355.04
	10240	187.33	253.33	1167.67	458.00	538.40	546.83	562.59	511.40	483.66	481.96	490.39	506.15	454.96	427.22
	12288	224.67	304.67	1371.33	537.67	616.21	626.33	645.24	583.82	550.53	566.41	576.54	595.45	534.02	500.73
	16384	299.00	403.00	1782.00	696.67	775.17	788.66	813.88	731.98	687.59	735.00	748.49	773.70	691.80	647.42

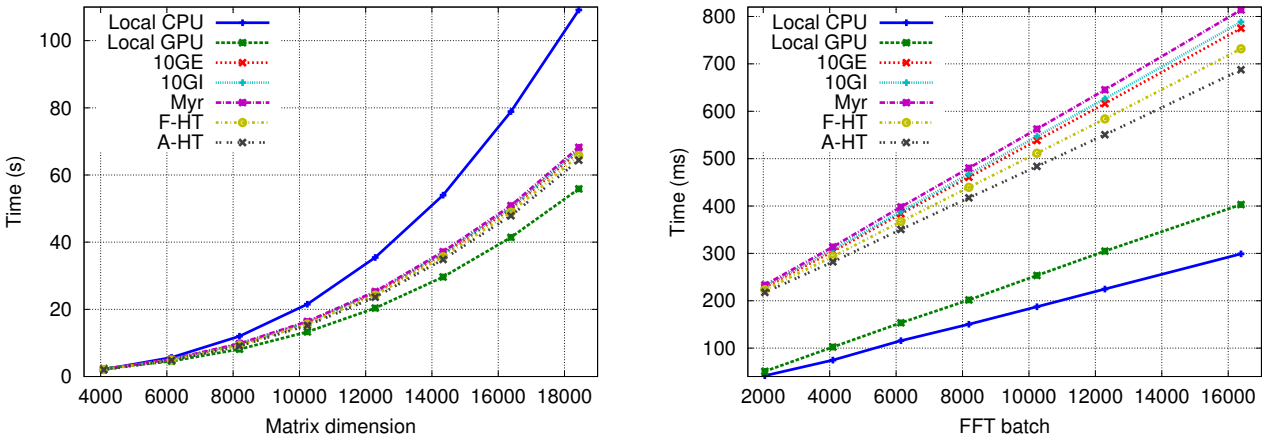


Figure 5. Processing times for the two case studies presented in this paper. **Left:** matrix product. **Right:** FFT. Estimated times based on the GigaE model.

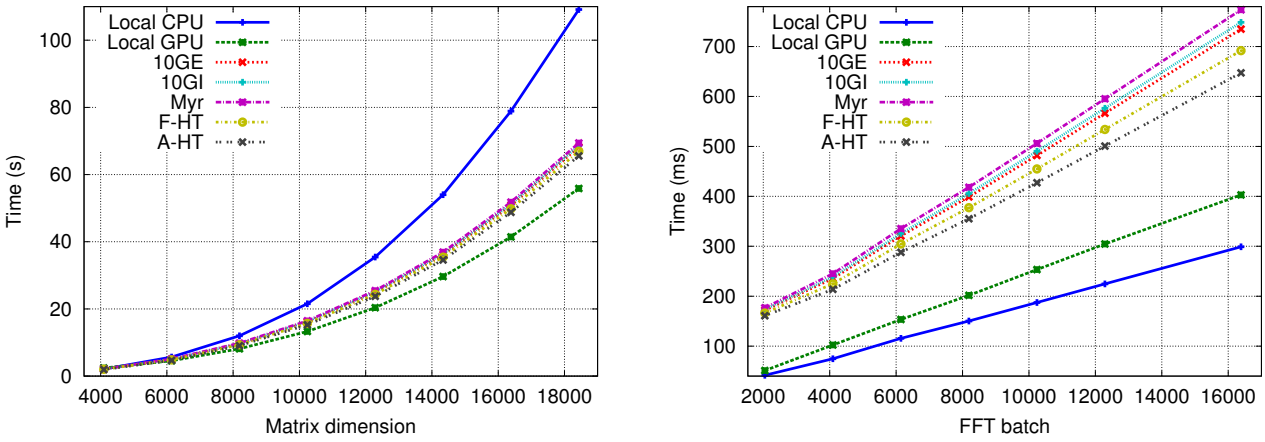


Figure 6. Processing times for the two case studies presented in this paper. **Left:** matrix product. **Right:** FFT. Estimated times based on the 40GI model.

variety of high performance networks. Our results show that for those cases that are eligible for being accelerated in a local GPU, we can estimate the performance of the remotely-accelerated solution within an acceptable accuracy, thus providing a tool to determine the behavior of our proposal over different interconnects with no need of the physical equipment. In this regard, we determined that employing a virtualized remote GPU is feasible if the problem is computationally intensive and, therefore, well suited to be accelerated by a GPU.

In future work we intend to extend our study to analyze the behavior of this proposal over a wide range of applications, cluster configurations, and network topologies, in order to be able to determine the exact amount of GPUs necessary in each particular case. Scheduling of multiple GPUs being simultaneously accessed by several applications also needs to be addressed.

Further Information

For further details on rCUDA, visit its web pages at UPV (http://www.gap.upv.es/rCUDA) or UJI (http://www.hpca.uji.es/rCUDA). In particular, these sites provide instructions on how to obtain a copy of rCUDA.

Acknowledgments

The researchers at UPV were supported by PROMETEO from Generalitat Valenciana (GVA) under Grant PROMETEO/2008/060, while those at UJI were supported by the Spanish Ministry of Science and FEDER (contract no. TIN2008-06570-C04), and by the Fundación Caixa-Castelló/ Bancaixa (no. P1-1B2009-35).

References

[1] W. Liu, B. Schmidt, G. Voss, A. Schroeder, and W. Muller-Wittig, "Bio-sequence database scanning on a GPU," in *IPDPS*, Rhodes Island, Apr. 2006.

[2] D. Shreiner and OpenGL, *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Versions 3.0 and 3.1 (7th Edition)*. Addison-Wesley Professional, Aug. 2009.

[3] F. D. Igual, R. Mayo, and E. S. Quintana-Ortí, "Attaining high performance in general-purpose computations on current graphics processors," in *High Performance Computing for Computational Science — VECPAR 2008*, ser. Lecture Notes in Computer Science, vol. 5336, pp. 406–419.

[4] W. R. Mark, S. R. Glanville, K. Akeley, and M. J. Kilgard, "Cg: a system for programming graphics hardware in a C-like language," in *SIGGRAPH '03*, New York, NY, USA, 2003.

[5] E. H. Phillips, Y. Zhang, R. L. Davis, and J. D. Owens, "Rapid aerodynamic performance prediction on a cluster of graphics processing units," in *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, no. AIAA 2009-565, Jan. 2009.

[6] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Ortí, and G. Quintana-Ortí, "Exploiting the capabilities of modern GPUs for dense matrix computations," *Concurr. Comput. : Pract. Exper.*, vol. 21, no. 18, 2009.

[7] J. Duato, F. D. Igual, R. Mayo, A. J. Peña, E. S. Quintana-Ortí, and F. Silla, "An efficient implementation of GPU virtualization in high performance clusters," in *Euro-Par 2009 Workshops*, ser. LNCS, vol. 6043, 2010, pp. 385–394.

[8] NextIO, "N2800-ICA — Flexible and manageable I/O expansion and virtualization." [Online]. Available: http://www.nextio.com/docs/NextIO%20N2800-ICA%20IO%20Consolidation%20Appliance%20Product%20Brief%20v0.18.pdf

[9] V. Krishnan, "Towards an integrated IO and clustering solution using PCI Express," in *2007 IEEE International Conference on Cluster Computing*, 2007, pp. 259–266.

[10] D. Blythe, "The Direct3D 10 system," *ACM Trans. Graph*, vol. 25, no. 3, pp. 724–734, 2006.

[11] *NVIDIA CUDA Reference Manual*. NVIDIA, 2010.

[12] A. Munshi, Ed., *OpenCL 1.0 Specification*. Khronos OpenCL Working Group, 2009.

[13] M. Dowty and J. Sugerman, "GPU virtualization on VMware's hosted I/O architecture," in *First Workshop on I/O Virtualization*. USENIX Association, Dec. 2008.

[14] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara, "VMM-independent graphics acceleration," in *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*. New York, NY, USA: ACM, 2007, pp. 33–43.

[15] L. Shi, H. Chen, and J. Sun, "vCUDA: GPU accelerated high performance computing in virtual machines," in *International Parallel & Distributed Processing Symposium*, 2009.

[16] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-accelerated virtual machines," in *3rd Workshop on System-level Virtualization for High Performance Computing*, 2009, pp. 17–24.

[17] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU transparent virtualization component for high performance computing clouds," in *Euro-Par 2010 - Parallel Processing*, ser. LNCS, 2010, vol. 6271, pp. 379–391.

[18] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh, "A package for OpenCL based heterogeneous computing on clusters with many GPU devices," in *Workshop on Parallel Programming and Applications on Accelerator Clusters*, Sep. 2010.

[19] J. Nagle, "Congestion control in IP/TCP internetworks," *Computer Communication Review*, vol. 14, no. 4, pp. 11–17, 1984.

[20] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Piscataway, NJ, USA, 2008.

[21] M. J. Rashti and A. Afsahi, "10-gigabit iWARP ethernet: Comparative performance analysis with InfiniBand and Myrinet-10G," in *CAC'07*, Long Beach, CA, USA, Mar. 2007.

[22] J. Duato, F. Silla, S. Yalamanchili, B. Holden, P. Miranda, J. Underhill, M. Cavalli, and U. Brüning, "Extending HyperTransport protocol for improved scalability," in *Proceedings of the First International Workshop on HyperTransport Research and Applications (WHTRA2009)*, 2009, pp. 46–53.