Universitat Politècnica de València

Master Thesis

Using remote accelerators to improve the performance of mathematical libraries

Author:
Santiago Mislata Valero

Advisor:
Federico Silla

A thesis submitted in fulfillment of the requirements
for the Master's Degree in Computer and Network Engineering

in the

Parallel Architectures Group
Department of Computer Engineering

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

February 2016

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Abstract

Department of Computer Engineering

Master's Degree in Computer and Network Engineering

Using remote accelerators to improve the performance of mathematical libraries

by Santiago Mislata Valero

Virtualization technologies, such as virtual machines, have demonstrated to provide economic savings to data centers because they increase overall resource utilization. However, in addition to virtualize entire computers, the virtualization of individual devices also provides significant benefits. As an example, the remote GPU virtualization technique moves local GPU computations to remote GPU devices.

Nevertheless, it is also possible to offload the computationally intensive CPU parts of an application to remote accelerators. In this work we present a first implementation of a new middleware that uses remote accelerators to perform the computations of scientific libraries, which were initially intended to be executed in the local CPU. Moreover, forwarding the computationally intensive parts of these libraries to remote accelerators is done in a transparent way to applications, not having to modify their source code. The first implementation of the new middleware is based on offloading the FFT and BLAS libraries, which are used to present the benefits of the new proposal by carrying out an in-depth performance evaluation. Results demonstrate that the new middleware is feasible. Moreover, scientific libraries such as FFTW may experience a speed-up larger than 25x, despite of having to transfer data back and forth to the remote server owning the accelerator.

Acknowledgements

En primer lugar, quería expresar mis agradecimientos a mi director, Federico Silla, ya que gracias a su experiencia, paciencia y comprensión han servido para que, por fin, pueda terminar esta tesina de máster que tanto tiempo estaba pendiente. Gracias a sus conocimientos en las tecnologías de virtualización y redes de interconexión de altas prestaciones, así como su asistencia a la hora de realizar documentos escritos (como papers, presentaciones o esta tesina), este trabajo pudo salir adelante. De igual manera, me gustaría agradecer también a José Duato, director del Grupo de Arquitecturas Paralelas (GAP), porque hayan vuelto a contar conmigo para formar parte de este grupo.

Igualmente, agradezco la ayuda y apoyo mostrado por los miembros del DISCA y compañeros del GAP.

Por supuesto, no voy a olvidarme de todos aquellos que siempre pensaron que terminaría esta tesina, aunque ahora no se crean que en realidad lo haya hecho: a mi familia y mis amigos de Utiel y Valencia, especialmente a Salva y Cris (sí, está terminada).

Contents

List of Figures

Acronyms

API	**A**pplication **P**rogramming **I**nterface
ARM	**A**dvanced **R**ISC **M**achine
BLAS	**B**asic **L**inear **A**lgebra **S**ubroutine
CPU	**C**entral **P**rocessing **U**nit
cuBLAS	**cu**DA **BLAS**
cuBLAS-XT	**cuBLAS** e**X**tended **T**echnology
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
cuFFT	**cu**DA **FFT**
DDR3	**D**ouble **D**ata **R**ate type **3**
DS-CUDA	**D**istributed-**S**hared **CUDA**
EDR	**E**nhanced **D**ata **R**ate
FDR	**F**ourteen **D**ata **R**ate
FFT	**F**ast **F**ourier **T**ransform
FFTW	**FFT** to the **W**est
GDDR5	**G**raphics **D**ouble **D**ata **R**ate type **5**
GPU	**G**raphics **P**rocessing **U**nit
IP	**I**nternet **P**rotocol
KVM	**K**ernel-based **V**irtual **M**achine
LAPACK	**L**inear **A**lgebra **PACK**age
MAGMA	**M**atrix **A**lgebra on **G**PU and **M**ulticore **A**rchitectures
MKL	**M**ath **K**ernel **L**ibrary
PCIe	**P**eripheral **C**omponent **I**nterconnect **e**xpress
PLASMA	**P**arallel **L**inear **A**lgebra for **S**calable **M**ulticore **A**rchitectures
QDR	**Q**uad **D**ata **R**ate
rCUDA	**r**emote **CUDA**
TCP	**T**ransmission **C**ontrol **P**rotocol

Chapter 1

Introduction

Virtualization technologies have attracted the attention of data center administrators due to the economic savings provided by these technologies: acquisition and maintenance costs are reduced as well as energy consumption requirements. In this regard, virtualization can be applied at different levels, as exposed below.

Firstly, virtualization can be applied at the computer level, leading to the widely used virtual machines. Examples of this technology are frameworks such as VMware [1], Xen [2], KVM [3], or VirtualBox [4], which have became so popular because several virtual machines can be concurrently executed in a real computer, sharing its resources and hence increasing overall resource utilization. As a consequence of the widespread use of virtual machines, processor manufacturers like Intel or AMD incorporate an increasing virtualization support into their processors [5].

Secondly, virtualization can also be applied at the device level in order to provide support to virtual machines. For instance, network adapters for technologies and manufacturers as different as Mellanox' InfiniBand or Intel's Ethernet include virtualization features [6][7], which basically allow the network adapter to be replicated, at the logic level, and assign different replicas to different virtual machines. Moreover, graphics processing units (GPUs) have also included virtualization support recently. This is the case for the GRID K1 GPU by NVIDIA [8], which can be shared among 8 virtual machines, or Intel GPUs, which incorporate virtualization support for the KVM framework [9]. It is worth mentioning that this support in GPU is mainly intended for desktop virtualization.

Notice, however, that in addition to provide support to virtual machines, virtualization of individual components of a computer may be also intended to provide an increased degree of flexibility at the cluster level. For example, networked disks enable to share a file system across a cluster. In a similar way, the recent remote GPU virtualization technique, implemented in frameworks like rCUDA [10], GVirtuS [11], or DS-CUDA [12], among others, allows a set of GPUs to be shared among several cluster nodes. Moreover, although remote GPU virtualization frameworks do not usually reduce execution time for single applications, when applied to a batch of computing jobs this technology allows reducing the total execution time [13] and the total energy consumed [14] by such aggregation of jobs.

Networked disks and remote GPU virtualization technologies are examples of the use of resources external to the computer that is executing the application using them. However, it is also possible to use external resources to carry out the computations initially devised to be performed by the local CPU, although in this case, not only an increased configuration flexibility is expected, as in the case of the remote GPU virtualization technique, but also a noticeable reduction in execution time should be achieved. In this thesis we introduce a new middleware that allows the computationally intensive CPU parts of an application to be offloaded to accelerators located in other nodes of the cluster. Furthermore, this outsourcing process is carried out in a completely transparent way to applications, without having to modify their source code.

The main contributions of this thesis are (1) a new middleware is introduced, (2) a first implementation of the new middleware, focused on the FFT and BLAS libraries, is presented, (3) some optimizations are thoroughly discussed, and (4) a comprehensive performance evaluation is carried out.

In order to properly present our proposal, the rest of the thesis is organized as follows. Chapter 2 presents in detail the general idea of the new middleware. Chapter 3 discusses the main implementation details of the new middleware for the FFTW and OpenBLAS libraries. The performance of the new middleware is later evaluated in Chapter 4 for each library by making use of real executions in several system configurations. Next, Chapter 5 presents a thorough performance estimation of the new middleware when its communication layer is optimized, as well as the benefits of applying some architectural optimizations. Chapter 6 further discusses some additional aspects of the proposal.

Later, the new middleware is compared against related proposals in Chapter 7. Finally, Chapter 8 presents the main conclusions.

Chapter 2

Offloading Local CPU Computations to Remote Accelerators

The idea of the new middleware is simple: it consists in moving 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 2.1. The client side is automatically contacted by the application as soon as it makes a call to one of the functions of the offloaded 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, LAPACK, FFT, 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

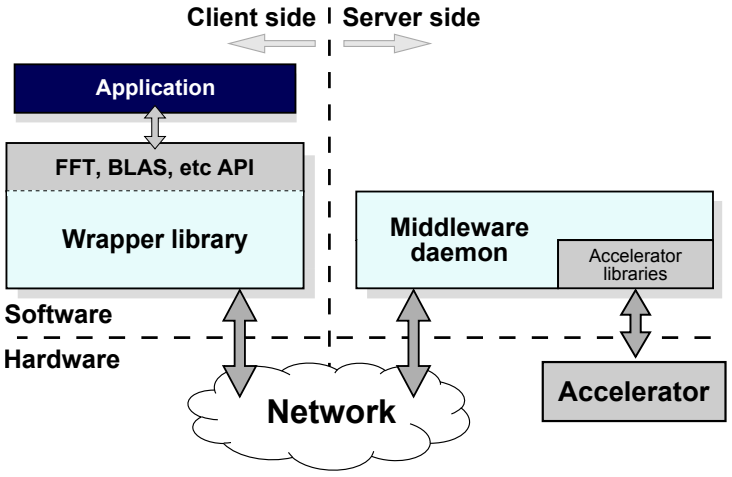


FIGURE 2.1: Architecture of the new middleware.

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 source 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 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

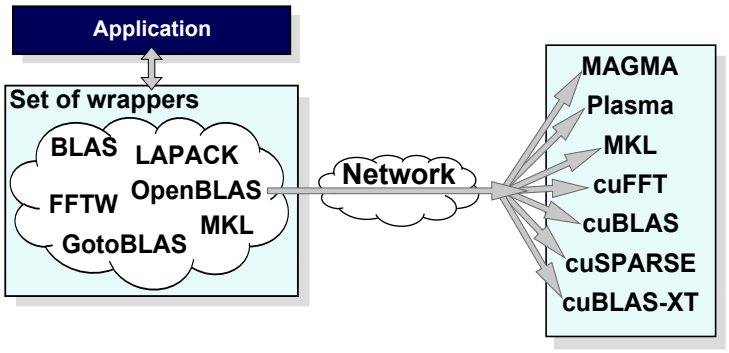


FIGURE 2.2: Architecture of the new middleware from a library perspective.

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 2.2 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 [15] library by NVIDIA might be leveraged to accelerate the computations of libraries such as GotoBLAS [16] or OpenBLAS [17]. Similarly, the MAGMA [18] and PLASMA [18] libraries could also be used. It is also possible that the server features several GPUs. In this case the cuBLAS-XT library [19] 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, the application would 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 [20], the NVIDIA cuFFT [21] 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 [22] 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 document, 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 FFT library is also implemented. Support for the LAPACK library will be addressed in future versions.

Chapter 3

Implementation Details of the New Middleware

Once the coarse details of the idea about the new middleware have been introduced, in this chapter we present how the new middleware has been implemented for the two mathematical libraries studied in this thesis (FFT and BLAS) when the remote server makes use of a CUDA-compatible GPU in order to accelerate computations. In addition to consider other computationally intensive libraries in future research, other implementation options for each of the libraries presented in this chapter could also be feasible.

An important issue that has to be considered while implementing the proposed middleware for a given mathematical (or computationally intensive) library is that two different approaches can be followed. In the first one, the client part of the middleware consists of a set of extremely simple wrappers for each of the functions in the library being remotely accelerated. These wrappers just forward the operation along with its input data to the remote server and wait for the results. Although this naive implementation may probably achieve execution time reductions in most cases, a better approach can be followed. In the second approach, the client part of the middleware also forwards to the remote server the operation to be performed along with its input data, but it does so in such a way that several optimizations can be applied in order to achieve larger speedups. Obviously, the server part of the middleware has to be accordingly designed so that it actively collaborates in these optimizations. Also, notice that these optimizations will probably be different not only for each targeted library but also for

each particular mathematical function remotely accelerated within each library and will also depend on the exact acceleration technology used in the remote server (GPU, Xeon Phi, etc).

3.1 Offloading the FFTW Library

In order to implement the FFT library within the proposed middleware, the FFTW [23] library may be selected, which is widely used due to its very good performance. Nevertheless, other libraries such as the Intel MKL could also be selected without significantly modifying the discussion in this section. The FFTW library is a relatively small package composed of about 40 functions. It computes the discrete Fourier transform (DFT) of complex and real data, and also the discrete Hartley transform (DHT) of real data. The current work is based on the DFT.

An important detail about the internals of FFTW is that it may use different algorithms to compute different DFT transforms. The specific algorithm to use is carefully selected depending on the exact underlying hardware, so that the performance of the DFT is maximized. Thus, computing a DFT is split into two phases: first, the FFTW planner analyzes the fastest way to perform the requested DFT in the current hardware taking also into account input data size. The planner generates a data structure, called *plan*, containing the result of this analysis. Afterwards, the plan is executed to apply the DFT to input data. As a consequence of this division of labor, functions within the FFTW library can be coarsely classified into two types: those for creating the plan (which might be represented by the `fftw_plan` function, although there are other flavors of this function depending on the exact data types used) and those for executing the plan (represented by the most general `fftw_execute` function). The typical program flow for computing the DFT is first calling the `fftw_plan` function to create the plan and then calling the `fftw_execute` function for executing it.

Dividing the work among creating the plan and executing it imposes an important concern when offloading the computations to be performed by the FFTW library to remote accelerators. Basically, the concern is how to efficiently deal with the two functions involved in the DFT transform (`fftw_plan` and `fftw_execute`) *without having to forward twice the input data to the remote server*. Notice that it cannot be assumed that both

functions will be sequentially called because of two reasons. First, it is possible for an application to create several plans for the same input data and finally select one of them according to some criteria of the application programmer. This is what the Gromacs application [24] does, for instance. Second, it is possible to have several sets of input data, with different size, and create the plans for these sets and execute them in many different interleaved combinations. Hence, the strategy used for offloading the computations of the FFTW library must be carefully designed.

A first detail to consider when designing the strategy to offload the FFTW library is that plans are computed for a specific hardware. Thus, they must be created in the remote server, according to its hardware features. Moreover, in case several plans are created by the application, if input data is sent to the remote server by the wrapper replacing the `fftw_plan` function, then special care should be taken so that input data is not sent several times.

A second detail to be taken into account is that, for some flavors of the `fftw_execute` function, it may just have a single input parameter: the plan. Thus, as we are creating a general strategy able to deal with all the different versions of the `fftw_plan` and `fftw_execute` functions, it seems to be necessary that the plan created at the server is returned back to the client so that it can be used as the input parameter to the wrapper that replaces the `fftw_execute` function, which should just forward it to the remote server, where it will be used to compute the DFT. Notice that in case several plans were created, the original code of the application will select the exact plan to use, according to the original programmer's directives, and will deliver such plan to the wrapper that replaces the `fftw_execute` function.

One more concern to be considered when designing the strategy to outsource the FFTW library is that, in the simplest version of the `fftw_execute` function, the memory pointers to input and output data used to compute the DFT transform are embedded into the plan, which, additionally, is an opaque data structure which does not allow retrieving the location of input and output data. Knowing the location of input data is not actually required given that the `fftw_execute` function at the server will use the plan and therefore will automatically access input data. However, it is necessary to know the location of output data in order to return it back to the client computer so that output data is

delivered there to the original application. Thus, the pointer to output data, which is known at the `fftw_plan` function, should be stored somewhere in the server.

Having all these concerns in mind, a feasible strategy to be followed to outsource the FFTW library could be to forward input data to the remote server when the `fftw_plan` function is called. Then, in addition to creating the plan, input data should be stored at the server as well as the pointer to output data. Once the plan is created, it should be returned back to the client, so that it is later used to feed the `fftw_execute` function wrapper. In case several plans are computed for the same input data, only the first call to the `fftw_plan` wrapper should forward data to the server. Moreover, when the `fftw_execute` function is called in the client machine, the plan is forwarded to the server and the DFT transform is computed there (without forwarding again input data). Finally, output data is returned to the client using the pointer previously stored.

Although feasible, the described approach to offload the FFTW library to a remote server presents two concerns. The first one is related to the possibility of computing several plans for different input data sets presenting different sizes. In this case, the aggregation of all input data, for the several plans, may probably not fit into GPU memory and therefore input data should be moved back and forth to the server main memory, what may translate into an important time overhead. One solution could be to store input data in main memory instead of in the GPU memory. However, this possibility leads us to the second concern: if data is stored in main memory at the server, then when the request from the `fftw_execute` wrapper is received, input data must be moved from main memory to GPU memory, requiring a non-negligible amount of time. Thus, a different strategy should be followed.

The approach finally followed for offloading the FFTW library consists in the wrapper of the `fftw_plan` function storing its input parameters in an internal data structure within the wrapper library at the client computer and not forwarding any information nor request to the remote server at that stage. Notice, however, that the `fftw_plan` wrapper must return a plan. As no plan has actually been created, the wrapper for the `fftw_plan` function will return a fake plan which has been initialized with an internal identifier, which will be later used by the `fftw_execute` wrapper to locate the stored parameters in the internal data structure of the wrapper library. In this way, when the `fftw_execute` function is called, the stored parameters are used to locate and forward the input data

to the remote server and store the results once received from the server. At the remote server, once input data is received, the plan will be computed before applying the DFT transform to the received data.

Notice that the proposed approach is compatible with the creation of several plans because the wrapper of the `fftw_plan` function stores in the internal data structure of the new middleware the parameters for all the different fake plans (with different identifiers, obviously). Therefore, when finally executing one of them, the wrapper for the `fftw_execute` function will browse the internal data structure searching for the exact plan to be executed and will forward the right parameters, along with the input data, to the remote server. Furthermore, contrary to what happened in the previous approach, the proposed strategy makes a very efficient use of the memory of the GPU at the server, since only the input data actually needed for the requested DFT transform will be stored at the GPU memory. In this regard, this second approach is simpler and more scalable.

One possible optimization to the presented approach is based on the fact that the exact value of input data is not required for creating a plan, but only input data size needs to be known. Thus, it is possible to enhance the previous strategy by sending a short command to the remote server as soon as the wrapper to the `fftw_plan` is called at the client, asking the server for creating the plan for such data size in advance (notice that the `fftw_plan` wrapper at the client will still return a fake plan, as described above. Furthermore, the server should allocate the required memory for input data before creating the plan, given that such pointer is needed). In this way, once input data is later forwarded by the `fftw_execute` wrapper, the plan will probably have been already created at the server and hence the total execution time will be accordingly reduced. This optimization requires to make sure that the plan is already created before actual computation of the DFT transform begins. However, checking this issue is straightforward and will generate no additional time overhead. On the contrary, the internal structure of the middleware server must be enhanced by making use of threads. A performance comparison between this strategy and the non-optimized one will be carried out in Chapter 5.

One final consideration about the implementation of the new middleware is how the DFT is computed at the remote server, which in this initial implementation owns a GPU. In this regard, once input data has arrived at the remote server, it is transferred to the GPU by making use of regular GPU memory copies (the GPU has been previous

initialized). After copying input data to GPU memory, the appropriate function of the cuFFT library by NVIDIA is used to compute the DFT in the GPU. Once the transform is completed, results are copied back from GPU memory to main memory in the server and then they are sent to the client side of the middleware. After receiving the results at the client side, the call that the application performed to the `fftw_execute` function is completed and application execution is resumed.

3.2 Offloading the OpenBLAS Library

In the same way as we selected the FFTW software as an example for offloading the FFT library, in the case for the BLAS library we have to select a target package. In this case 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 [17]. However, as it happened with the FFTW library in the previous section, selecting a different candidate for the BLAS outsourcing would not significantly modify the discussion in this chapter.

In order to present the main idea about the offloading process of the BLAS library, we may start with a simple analysis of three representative functions of the BLAS library: DGEMM, DGEMV, and DDOT. These three functions, which use double precision data, 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. 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.

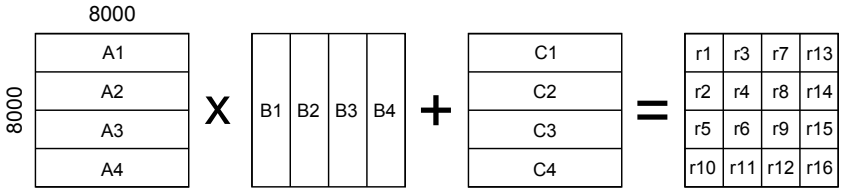
Another interesting point of view when analyzing these functions is the amount of computations performed per input data element. Assuming the use of square matrices in order to simplify this analysis, the DGEMM function requires $3n^2$ input elements, being

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 if only multiplications are considered. As can be seen, the different computational 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.

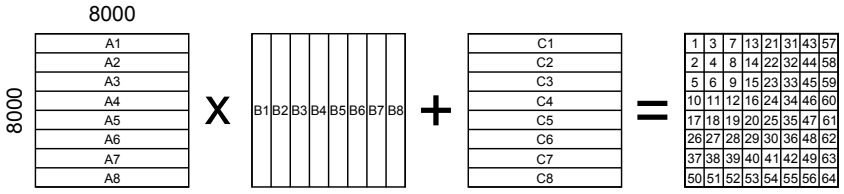
Given that the main concern in the proposed middleware is moving data to the remote server, it can be easily derived that the functions of the BLAS library that best fit into the idea of the new middleware are those belonging to the Level 3 of BLAS. This is why we are going to discuss first some ideas about their implementation, using the DGEMM function as an example. Notice, however, that other BLAS3 functions should probably undergo a different discussion specially tuned for the particular operation they perform.

As it happened to the FFT implementation discussed in the previous section, in the case of the DGEMM function one may think about two possible ways of outsourcing its computations. 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. This is the approach we follow for the first version of the middleware.

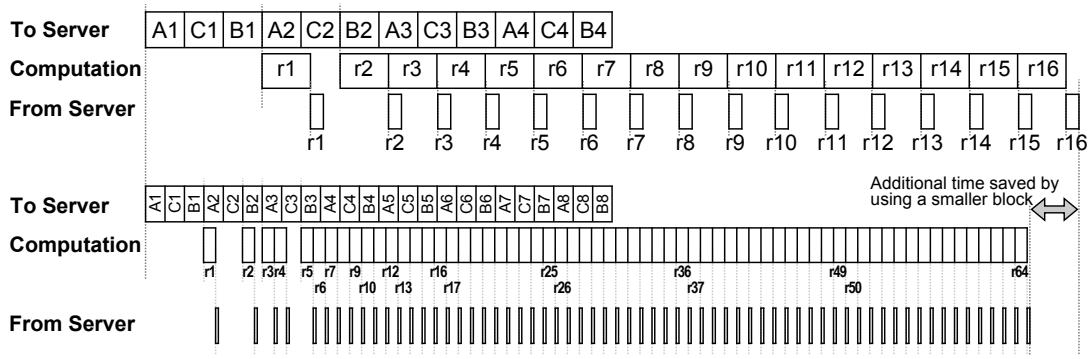
Although this naive implementation would probably provide some acceleration with respect to performing the DGEMM function in the local CPU cores, there is a smarter way to carry out the offloading process, which is based on combining data transmission with computations in a pipelined way. Figure 3.1 describes this process as an optimization of the middleware with a matrix example.



(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 the DGEMM operation when matrices are split into four and eight blocks.

FIGURE 3.1: Pipelining the DGEMM operation.

Figure 3.1(a) 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 3.1(a) 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 drawback of the proposed middleware. This is shown in the upper part of Figure 3.1(c). 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[1]. In a similar way, once

[1]It is important to remark that the width of the boxes in Figure 3.1 is proportioned to their actual duration in the experimental setup used in Chapters 4 and 5.

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 3.1(b) 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 3.1(c) 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. Moreover, it is very common that network fabrics attain their maximum bandwidth for transfer sizes larger than some minimum length. Thus, the use of very small blocks would allow to begin computations in the remote GPU earlier but would probably present some drawbacks that may probably reduce the overall benefits. Analyzing the optimal block size is an open issue that mainly depends on the specific GPU used at the remote server and also on the exact network fabric used.

Finally, the discussion above was focused on the DGEMM operation. In a similar way, an analysis for the DGEMV and DDOT functions should be performed with both the naive implementation and the pipeline optimization. However, given the low computation-to-communication ratio that these operations present, it is very likely that applying the

pipelining optimization to them provides no real acceleration with respect to carrying them out with the naive approach.

Chapter 4

Performance Evaluation

In this chapter we present a thorough performance evaluation of the new middleware based on real executions. With respect to the testbed used in the experiments, the client node executing the original application will be either an Atom or Xeon-based system. Notice that the reason for including Atom systems in this performance evaluation is based on the recent proposals that consider low-power processors such as Atoms or ARMs to be used in datacenters. The X10 MicroBlade server by Supermicro [25], which includes up to 6272 Avoton cores (architecture Atom), is an example of this trend. A system with an ARM architecture has also been included in the study, but due to some technical limitations with it (few memory, only Ethernet connector), the results obtained will be shown in Appendix C.

The exact characteristics of the systems used in this study are the following. The Atom-based system is a Supermicro 5018A-TN4 server containing an 8-core Atom C2750 processor working at 2.4 GHz and 16 GB of memory at 1600 MHz. It includes one PCIe 2.0 x8 connector. The Xeon-based system is a 1027GR-TRF Supermicro server featuring two Intel Xeon hexa-core E5-2620v2 processors (Ivy Bridge) operating at 2.14 GHz and 32 GB of memory at 1600 MHz. An additional Xeon-based system has been considered, for comparison purposes, comprising two Intel Xeon quad-core E5-2637v2 processors (Ivy Bridge) working at 3.5 GHz and 32 GB of memory at 1866 MHz.

For the server part, the one that actually performs the computations, we considered a 1027GR-TRF Supermicro server featuring two 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 along with Mellanox OFED 2.3-2.0.0 (InfiniBand drivers and administrative tools) and CUDA 6.5 with NVIDIA driver 340.29. The same software configuration (except for the NVIDIA software) was used in the client nodes.

With respect to the network fabric connecting the client and server nodes we have considered, for the Xeon-based systems, 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 has been also used. In the case for the Atom-based system, only Ethernet and InfiniBand QDR were used due to the lack of a PCIe 3.0 connector in this system. Furthermore, notice that TCP/IP communications will be 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 data involved in the requested computation has arrived at the server, it is 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 overhead.

4.1 Performance of the FFTW Offloading

Figure 4.1 depicts the performance attained by the new framework when computing the one-dimensional DFT, using double complex data type (16 bytes) as input and output (complex-to-complex, C2C). The transform of the function involving real data (8 bytes) in the input (real-to-complex transform, R2C) or in the output (complex-to-real transform, C2R) was also considered, although similar results were obtained. These additional results are shown in Appendix A. Execution time of the original (non-outsourced) executions is included as reference. In this case, the system with Xeons at 2.1 GHz and the system with Xeons at 3.5 GHz are leveraged. Curves labeled *"Xeon 2.1 Local"* and *"Xeon 3.5 Local"* depict these results. The performance of the cuFFT executions in the GPU (without the new middleware) is also shown for comparison purposes (curve *"GPU Local"*). Notice that times shown for the *"GPU Local"* curve

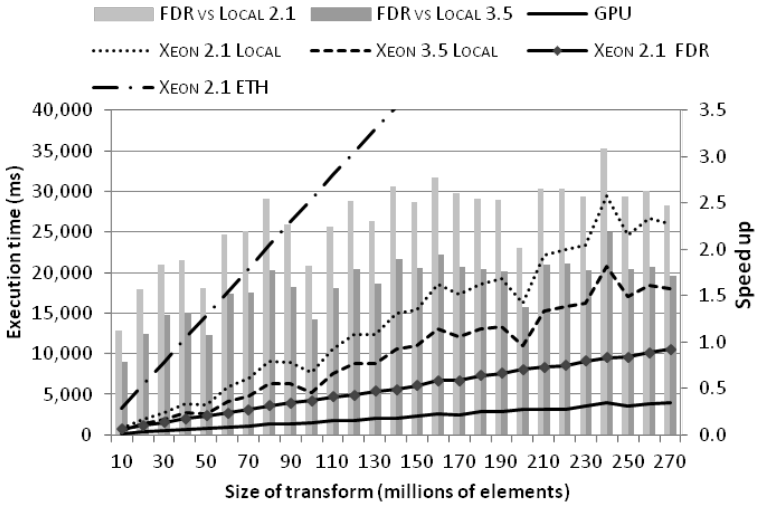
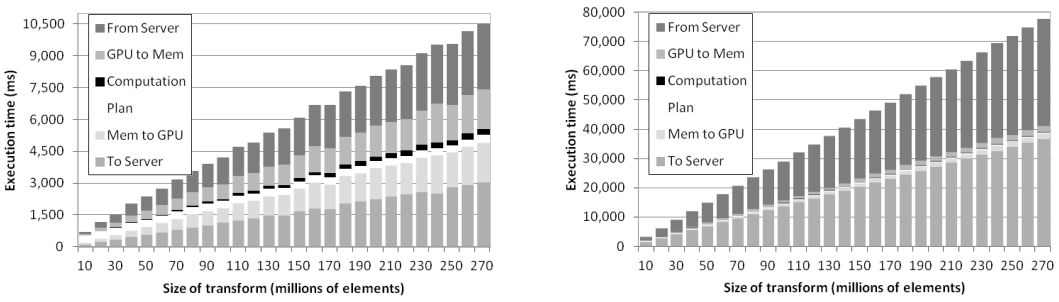


Figure 4.1: Performance of the C2C-DFT 1D function in the Xeon-based system.

include the creation of the plan as well as moving data between main memory and GPU memory. On the other hand, when the new middleware is used, the client node is a Xeon-based system operating at 2.1 GHz. The Ethernet network (curve labeled *"Xeon 2.1 ETH"*) as well as the Infiniband QDR and FDR network fabrics are considered. The speed up obtained when using the FDR interconnect is also shown. Bars labeled *"FDR vs Local 2.1"* and *"FDR vs Local 3.5"* refer, respectively, to the speed up with respect to the local execution in the Xeon 2.1 GHz and Xeon 3.5 GHz processors.

Notice that the results for the FDR fabric are displayed in the curve *"Xeon 2.1 FDR"*, but there is no information regarding the results for the QDR fabric. The reason why this curve has not been displayed in Figure 4.1 is because Infiniband QDR and FDR provided very similar results. As obtaining similar results for both networks was surprising, we decided to use the well known iperf tool [26] to gather TCP bandwidth results (remember that we are making use of TCP/IP over InfiniBand). This tool showed that for the Xeon 2.1 GHz system both interconnects provide basically the same performance, around 1190 MB/s. Interestingly, when the processors are upgraded to the 3.5 GHz version, the iperf tool reported noticeable better performance: 1883 MB/s for the QDR fabric and 2255 MB/s for the FDR network.

Figure 4.1 clearly shows that the use of a slower network, like 1 Gbps Ethernet, does not provide performance gains, despite of accelerating the computation of the DFT by using a powerful GPU. However, a faster network, such as InfiniBand, provides near 3x better execution times with respect to the Xeon 2.1 GHz and 1.75x better times



(a) InfiniBand FDR network adapter.

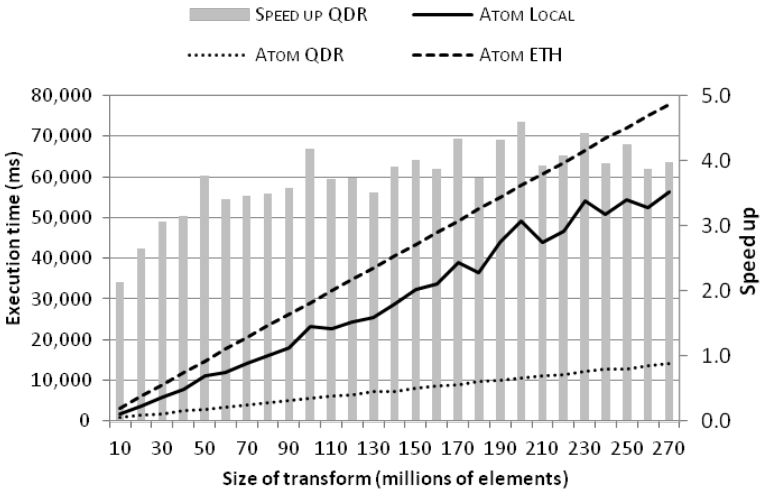
(b) ETH network adapter.

FIGURE 4.2: Execution time breakdown for the C2C-DFT 1D function in the Xeon-based system.

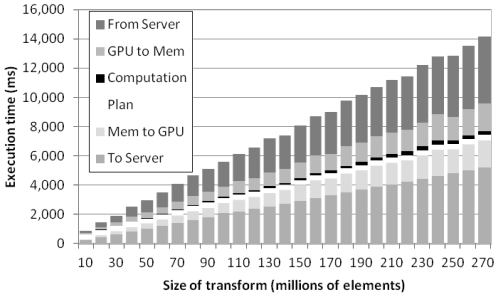
when the reference is the Xeon 3.5 GHz. Nevertheless, these accelerations are still far away from the maximum acceleration provided by the cuFFT library used with the local GPU (curve *"GPU Local"*). In this case, the local GPU provides a speed up between 30x and 35x (speed up not shown) with respect to executions of the FFTW library in the CPU. These results confirm that, in order to accelerate the executions of the FFTW library, network performance is a key concern, as expected. In next chapter we will see how by using an improved communication architecture over the same network hardware, the performance of the new middleware is noticeably increased. Finally, in all the experiments in Figure 4.1 the overhead introduced by the execution of the code of the new middleware is less than one millisecond.

Figure 4.2 presents a breakdown of the execution time when using the new middleware. Execution time is split into six components: (1) time required to move input data of the FFTW function from main memory in the client to main memory in the remote server (*"To Server"*), (2) time spent in the server to copy input data from main memory to GPU memory (*"Mem to GPU"*), (3) time required to complete the creation of the plan at the server (*"Plan"*), (4) time spent by the GPU to compute the DFT transform (*"Computation"*), (5) time required to move the results from GPU memory to main memory in the server (*"GPU To Mem"*), and (6) time for returning the results back to the client (*"From Server"*).

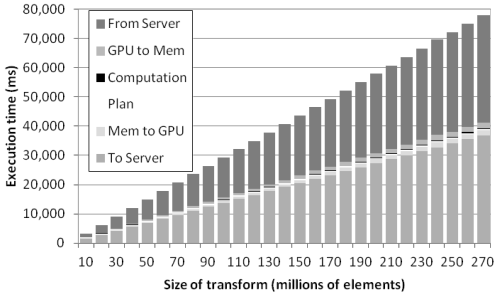
Figure 4.2(a) shows the breakdown using the InfiniBand FDR network adapter. Almost 60% of the time is spent in moving data between the client and the server computers, whereas about 37% of the time is used for moving data between main memory in the server and the GPU. Actual computation requires only 3% of the total time. When the Ethernet network is used (Figure 4.2(b)) about 95% of the time is spent in moving data



(a) Execution time and performance.



(b) Breakdown with IB QDR network adapter.

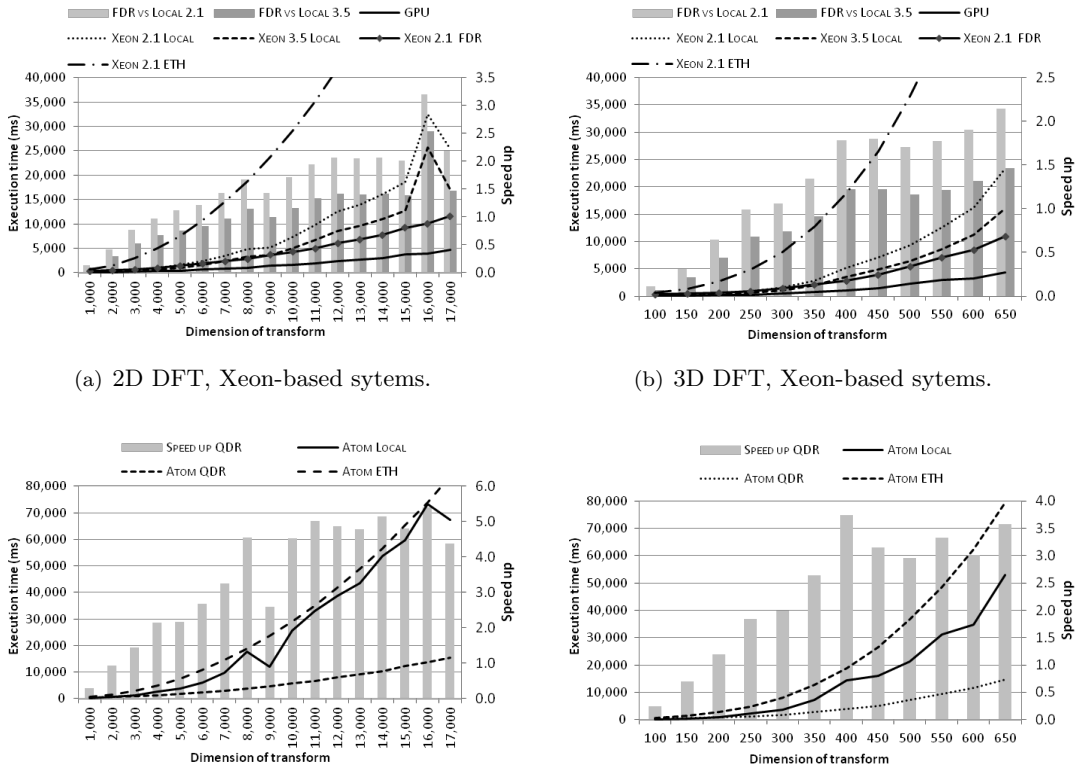


(c) Breakdown with ETH network adapter.

Figure 4.3: C2C-DFT 1D function in the Atom-based system.

to and from the remote server. This data movement time is further increased by the time required to move data between main memory in the server and the GPU, what finally causes that less than 0.4% of the total time is devoted to computations in the GPU. It will be shown in next chapter that a better communication architecture noticeably increases performance gains. Finally, notice that in both Figures 4.2(a) and 4.2(b) the creation of the plan is hidden for all transform sizes except for the smallest one using the InfiniBand FDR adapter, where the time for creating the plan partially overlaps with communication time.

The results regarding the use of our middleware in the Atom-based system are shown in Figure 4.3. Figure 4.3(a) depicts the execution time when the Ethernet and InfiniBand QDR fabrics are used. It can be seen that in this case, and due to the smaller computing capacity of the Atom processor with respect to the Xeon one, the speed up attained is larger. Figures 4.3(b) and 4.3(c) show the breakdown of the execution time considering the InfiniBand QDR adapter and Ethernet network, respectively. The results depicted



(a) 2D DFT, Xeon-based sytems.

(b) 3D DFT, Xeon-based sytems.

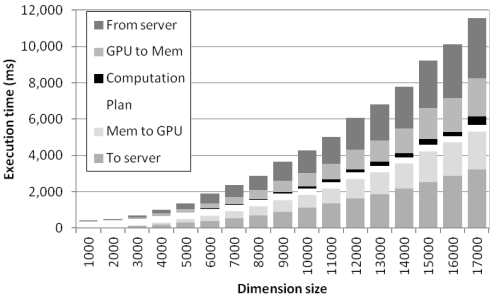
(c) 2D DFT, Atom-based systems.

(d) 3D DFT, Atom-based systems.

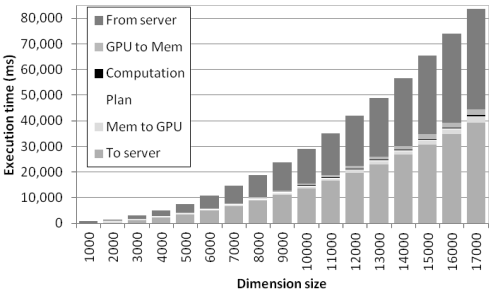
FIGURE 4.4: Performance of the C2C-DFT multi-dimensional functions.

are similar than the ones showed for the Xeon-based systems, being in this case the percentage of the time required for the computations smaller than the previous ones: around 2% of the total time when the InfiniBand QDR adapter is used, and 0.3% with the Ethernet network.

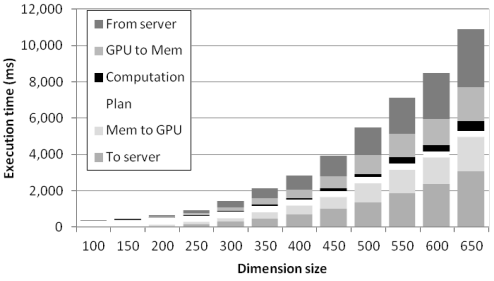
In the case of outsourcing multi-dimensional DFT transforms, Figure 4.4 shows execution time and speed up for the 2D and 3D transforms in Xeon and Atom-based systems, respectively. Input data size presents the same number of elements in the two or three dimensions. Experiments with input data having different number of elements in different dimensions provided similar results. Executions have been carried out in the same scenarios considered in Figures 4.1 and 4.3. Figure 4.4 shows that the use of the new middleware for accelerating the multi-dimensional DFT transform follows, in general, the same trend already analyzed for the 1D transform. In this regard, the use of a low performance network provides a noticeable degradation of execution time. On the contrary, the use of the high performance InfiniBand interconnect reduces total execution time. In the case of the 2D transform, between 1.5x and 2.5x speed up is achieved when compared to local executions in a Xeon 2.1 GHz. In the case of comparing against a



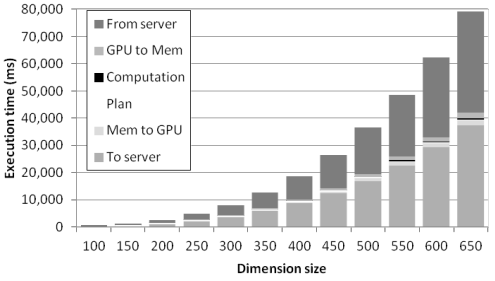
(a) 2D, IB-FDR, Xeon-based systems.



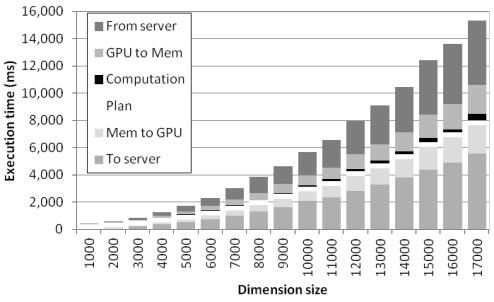
(b) 2D, Ethernet, Xeon-based systems.



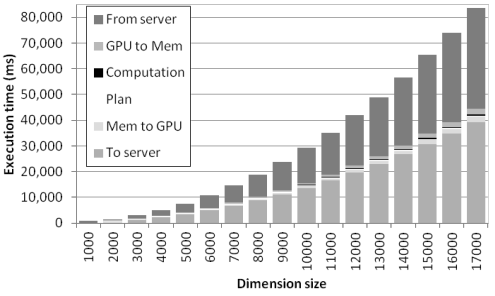
(c) 3D, IB-FDR, Xeon-based systems.



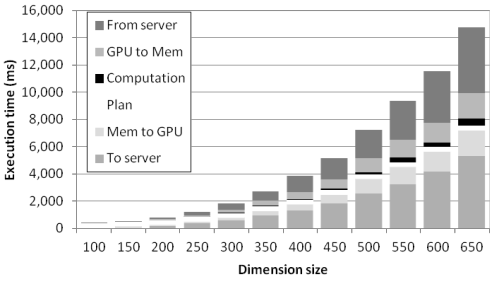
(d) 3D, Ethernet, Xeon-based systems.



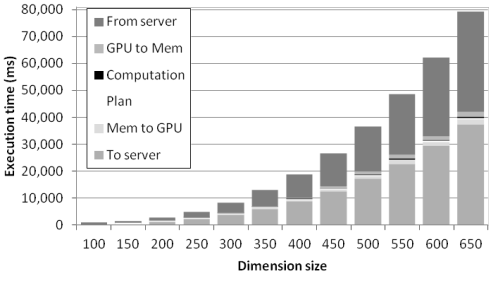
(e) 2D, IB-QDR, Atom-based systems.



(f) 2D, Ethernet, Atom-based systems.



(g) 3D, IB-QDR, Atom-based systems.



(h) 3D, Ethernet, Atom-based systems.

FIGURE 4.5: Breakdown of C2C-DFT multi-dimensional.

Xeon 3.5 GHz, attained speed up is around 1.5x. Similar results are obtained for the 3D transform, although speed ups are slightly lower: between 1.5x and 2x with respect to using a Xeon 2.1 GHz and around 1.25x when compared to local executions in a

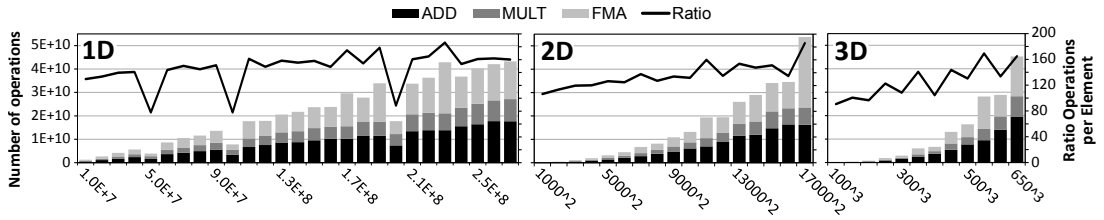


FIGURE 4.6: Amount of floating-point operations carried out in the 1D, 2D, and 3D FFTW transforms.

Xeon 3.5 GHz. When the client computer uses an ATOM processor, maximum speed up is 4x and 3x for the 2D and 3D DFT transforms, respectively. The breakdown of the execution time for both dimensions and systems, whose results are similar to the ones shown in Figures 4.2(a), 4.2(b), 4.3(b) and 4.3(c) for the 1D-DFT, are depicted in Figure 4.5.

Contrary to the trend depicted in Figure 4.1, where some speed up was attained for all the considered transform sizes, Figure 4.4 shows that in the case of the 2D and 3D transforms the smallest transform sizes report no performance gain. For instance, for the 2D DFT, transform sizes up to 3000^2 elements provide no acceleration with respect to local executions in CPU. In the case of the 3D DFT, transform sizes up to 200^3 elements experienced no acceleration. From these sizes, speed up starts growing as the number of elements increases, until it gets a steady state.

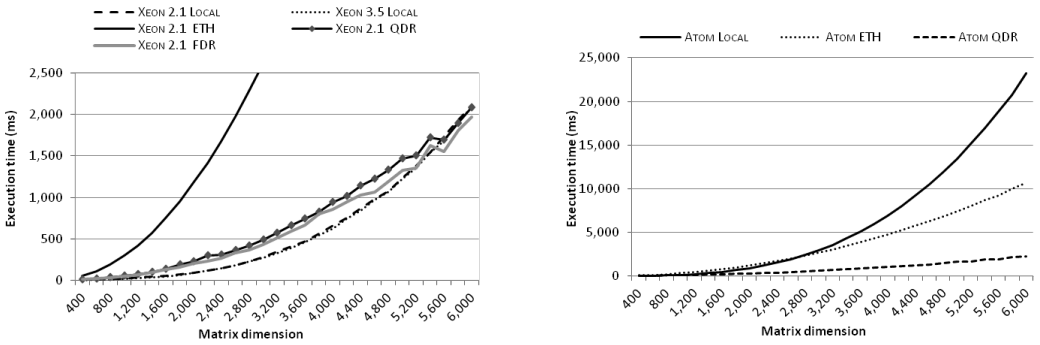
The reason for this different behavior among the one-dimensional transform and the multi-dimensional DFT can be found in Figure 4.6, which depicts the amount of floating point operations for each of these transforms. The amount of operations carried out for each transform size is broken down into the three operations used: additions (ADD), multiplications (MULT), and fused multiply-additions (FMA). These latter operations are composed of a multiplication and an addition performed in a single step in the GPU. Numbers in Figure 4.6 have been gathered by making use of a profiling feature already included in the FFTW library. As can be seen in Figure 4.6, the total amount of operations increases with transform size, as expected. However, what is more interesting is to analyze the ratio between the number of elements and the amount of operations, which is shown in the figure with the black line labeled *"Ratio"*. This ratio is less dependent on the transform size and points out that, for the smallest transform sizes, the amount of operations per element decreases as the amount of dimensions considered for the DFT transform increases. On the contrary, for the largest transform sizes, the

1D, 2D, and 3D transforms present similar ratios. Interestingly, smaller transform sizes do not experience any performance gain until the ratio reaches a value equal to or larger than 120, approximately, when the client node is a Xeon 2.1 GHz. In this regard, in the case of the 1D transform, the smallest transform size presents a ratio equal to 129, thus providing performance gains. Notice that this result is aligned with the general spirit of the new middleware, which accelerates computations at the cost of moving data to/from the remote server. Hence, the performance gains of the new middleware might be seen as a tradeoff between the time saved because of accelerating the computations with a GPU and the extra time spent in moving data to/from that remote GPU. This effect of the ratio among computations and communications will be more noticeable in next chapter, when a better communication layer is used.

4.2 Performance of the OpenBLAS Offloading

In the same way as we have done with the FFTW library, we have carried out a performance evaluation of the OpenBLAS library when offloading its representative functions (DGEMM, DGEMV and DDOT) to a remote GPU.

Figure 4.7(a) depicts the performance attained by the DGEMM function in Xeon-based systems. Execution times when using the local CPU have been included as references, and they are referred to by the curve labeled *"Xeon 2.1 Local"* and *"Xeon 3.5 Local"*, depending if the execution has been carried out in the system with the Xeon at 2.1 GHz or in the system with the Xeon at 3.5 GHz, respectively. Notice that the 12 available

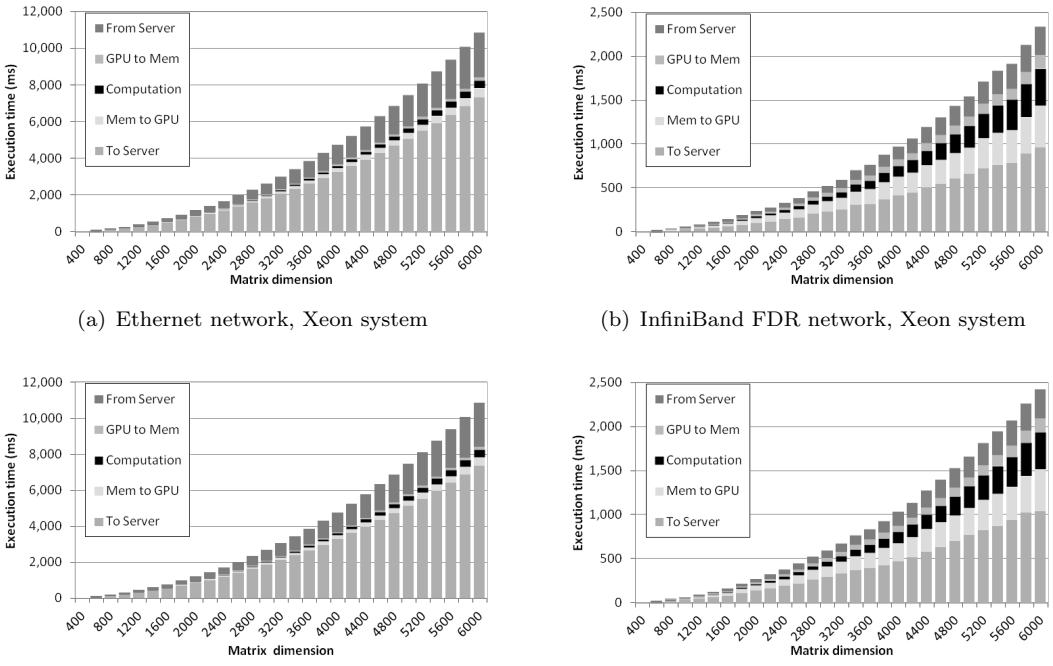


(a) Xeon-based systems.

(b) Atom-based systems.

Figure 4.7: Execution time of DGEMM function.

CPU cores in the system have been used in the local experiments for the Xeon at 2.1 GHz. The same experiments have been done with the system with Xeon at 3.5 GHz, using its 8 available cores. Figure 4.7(a) 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 clearly show, as in the case for the FFTW library, 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 chapter), the new middleware may provide some performance gains. Notice the similar results obtained for both QDR and FDR network, whose behavior has been already explained in the Chapter 4.1. Also remember that these results do not include the pipelined approach described in the previous chapter.



(a) Ethernet network, Xeon system

(b) InfiniBand FDR network, Xeon system

(c) Ethernet network, Atom system

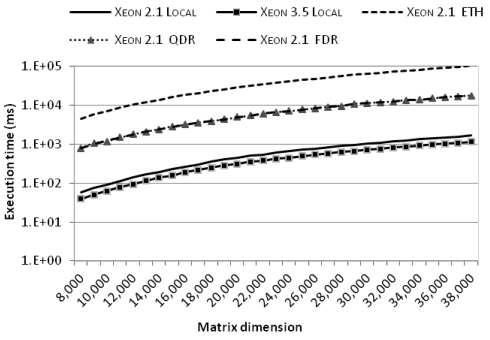
(d) InfiniBand QDR network, Atom system

FIGURE 4.8: DGEMM execution time breakdown.

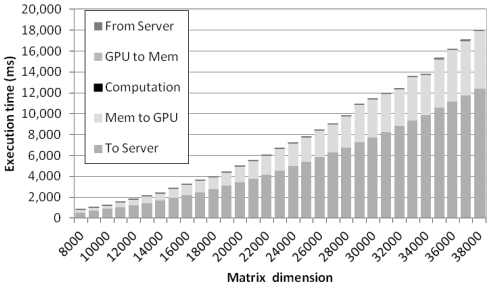
In the same way, Figure 4.7(b) shows the performance for the DGEMM function in Atom-based systems. In this case, the curve labeled *"Atom Local"* refers to the execution time of the function using the local CPU in the the Atom system. Notice that the 8 CPU cores have been used for the local experiments. In addition, curves labeled *"Atom ETH"* and *"Atom QDR"* show the execution time when our middleware is used along with an Ethernet and the InfiniBand QDR network fabric, respectively. In this case, the smaller computing capacity of the Atom processor compared to the Xeon one makes that, even moving the data to the server, using the new middleware with the Ethernet network is faster than making the executions locally.

Figure 4.8 presents the breakdown of the execution time when using the new middleware. Time is broken down into five components: (1) time required to move the input data of the DGEMM function to the main memory of the remote server, (2) time required to move the input data from the main memory of the server to the GPU memory, (3) time spent in the CPU doing the calculations, (4) time required to return back the results from the GPU memory to the main memory of the server, and (5) time required to return back the results to the client node. Labels *"To Server"*, *"Mem to GPU"*, *"Computation"*, *"GPU to Mem"*, and *"From Server"* refer, respectively, to each of these components. It can be seen that when using the Ethernet network in the Xeon-based scenario (Figure 4.8(a)), most of the total time is spent in moving data to/from the remote server (around 95%), 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 4.8(b)), although communications still represent an important fraction of the total time (nearly 60%), thus increasing total execution time with respect to local executions in the CPU cores, as shown in Figure 4.7(a). Breakdown of results for the Atom-based system (Figure 4.8(d) and 4.8(c)) show similar percentages although in this case some performance gains are attained.

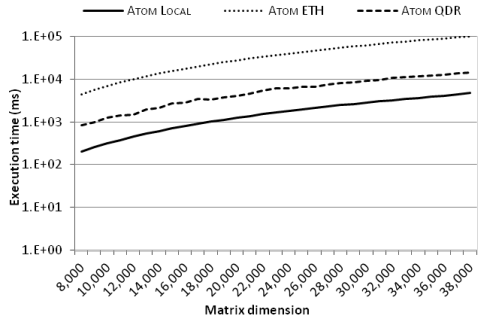
Contrary to the results of the DGEMM function, Figure 4.9(a) shows that for the DGEMV function there is no benefit on offloading the computations when using the TCP/IP protocol stack. Even for the faster InfiniBand interconnect, the use of our middleware increases execution time by a factor of 10x aproximately. The reason is that the penalty of moving data to the remote server is too large (around 70% of the total time when using InfiniBand, as shown in Figure 4.9(b)) and the faster computation of



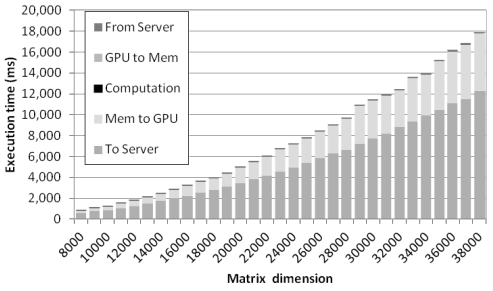
(a) Execution time in Xeon-based system.



(b) Breakdown with InfiniBand FDR.



(c) Execution time in Atom-based system.

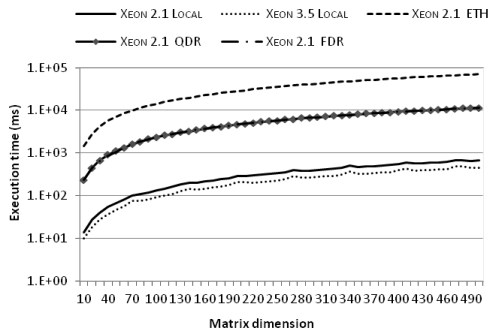


(d) Breakdown with InfiniBand QDR.

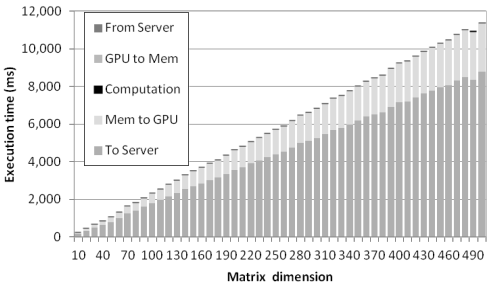
FIGURE 4.9: Performance of DGEMV function.

the Testa K40 GPU does not compensate for it. Experiments with the low-power processors provided similar results (Figure 4.9(c) and 4.9(d)). The smaller computation-to-communication ratio of this function is the responsible for no experiencing any execution time reduction.

Finally, Figure 4.10 shows the execution time and its breakdown for the DDOT function. In this case, and given the lower computational complexity of the DDOT function with



(a) Execution time in Xeon-based system.



(b) Breakdown with InfiniBand FDR.

FIGURE 4.10: Performance of DDOT function.

respect to the DGEMM and DGEMV functions, it can be easily understood that this function should not be offloaded, at least with the communication layer currently used in our middleware, which causes that sending the input data to the server required up to 75% of the total time.

Chapter 5

Performance Estimation With Improved Communications

In the previous chapter we have presented the performance attained by the initial implementation 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, it is moved to the GPU memory and then computation starts. However, it is possible to optimize such data movement in order to noticeably increase its throughput. For example, the InfiniBand Verbs API may be used instead of the TCP/IP protocol stack, boosting attained bandwidth. Also, an efficient communication pipeline could be leveraged, as in the rCUDA remote GPU virtualization framework [27]. Another possibility is using the GPU Direct RDMA mechanism provided by NVIDIA and Mellanox [28] when finally it is properly implemented and tuned so that it provides high performance. Moreover, notice that in these latter options, data is directly moved from main memory in the client node to GPU memory in the remote server, thus not only making use of a higher network bandwidth, but also of an improved communication architecture. Actually, avoiding the intermediate stop at the server's main memory is what really will improve performance.

Given that moving data is the main concern in the new framework, in this chapter we present a performance estimation if an optimized communication layer were used. The performance estimation methodology consists in replacing, in the results presented in

the previous chapter, the data transfer time from main memory in the client to 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 to move data directly between memory in the client node and the GPU memory in the server. 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 considered instead of using the raw bandwidth of the InfiniBand fabric. This approach is more accurate than using the raw network bandwidth because software layers always impose some loss to theoretical performance numbers. Additionally, notice that we have considered in this chapter the use of InfiniBand FDR (56 Gbps) and InfiniBand EDR (100 Gbps) network adapters.

5.1 Improved FFTW

In this chapter, it is presented a performance estimation of the execution time for the FFTW library. It has been considered the improved communication architecture previously explained. The optimization described in the previous chapter regarding plan overlapping has not been included in the first plots of this section. The estimation of the execution time including both the improved communication architecture and this optimization is shown in Appendix D.

Figure 5.1 presents the performance estimation of the new middleware. The figure shows, as the reference, the execution time of the 1D DFT using double complex data type as input and output (complex-to-complex transform) when carried out in the traditional way (using the local CPU). As mentioned in Chapter 4, results regarding real data, either in the input (real-to-complex transform) or the output (complex-to-real transform) show similar results, and they are shown in Appendix B. Curves *"Xeon"* and *"Atom"* refer, respectively, to the FFTW executions using the local Xeon 2.1 GHz and the Atom processors. When the new middleware makes use of the improved communication layer, curves *"Atom QDR"*, *"Xeon FDR"*, and *"Xeon EDR"* refer, respectively, to the FFTW executions using the remote accelerator from the Atom system with InfiniBand QDR, from the Xeon 2.1 GHz with InfiniBand FDR, and from the Xeon 2.1 GHz system with

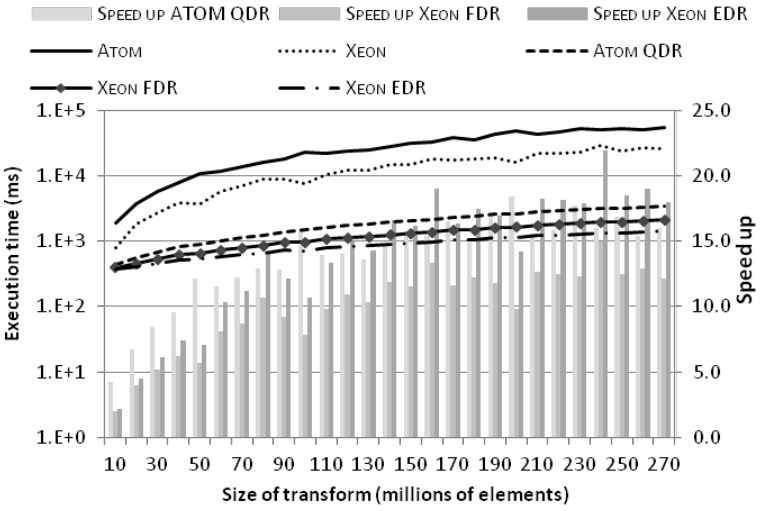


Figure 5.1: Estimation of the FFTW 1D in different system configurations.

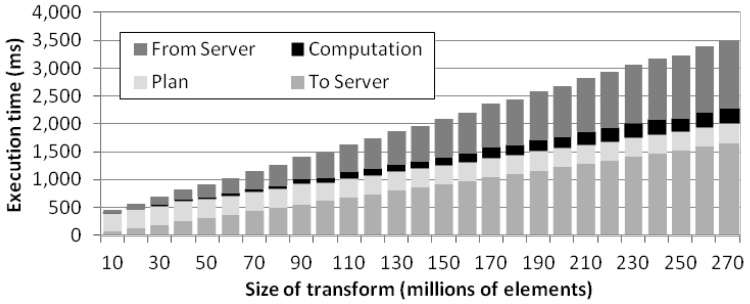
InfiniBand EDR. The speed up with respect to local executions is also shown. Notice the logarithmic scale in the execution time axis.

Figure 5.1 shows that when the main concern in the new middleware (data transfers across the network) is addressed by making use of an improved communication architecture, noticeable reductions in execution time can be achieved. For example, for transform sizes larger than 90 millions of elements, the use of the new middleware reports, in general, a speed up higher than 10x (or 15x) for InfiniBand FDR (or EDR). More specifically, when the InfiniBand EDR fabric is used, speed up is well above 15x. Notice, however, that in the case for the Atom system using InfiniBand QDR, for instance, speed up has been increased from 4x up to 17x, approximately, while communication bandwidth has been increased from 1.8 GB/s with TCP up to 3.3 GB/s with the improved communication architecture. Hence, how is it possible that FFTW has been sped up by a factor of 4.25x whereas communication bandwidth has only been increased by a factor of 2x? The answer is based on the fact that when TCP-based communications were used in the previous chapter, data 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 latency. 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. 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

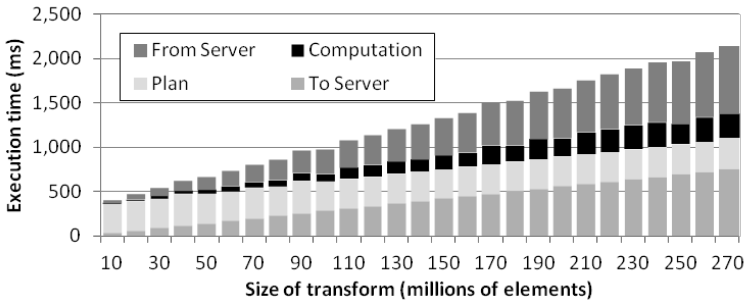
also due to the fact that data transmission is not following a stop&wait approach but it is done in a cut-through way, thus also reducing communication latency. Nevertheless, although an improved communication layer is being used, maximum speed up is still smaller than the provided by the cuFFT library (30x-35x) when the new middleware is not used.

Figures 5.2(a), 5.2(b), and 5.2(c) show the breakdown of the total estimated execution time. Notice that in this case the time breakdown is slightly different from the one depicted in the previous chapter, given that in this case data is directly transferred between main memory in the client and GPU memory in the server without being stored in the server's main memory. In this way, time *"To Server"* refers to the movement of input data from main memory in the client node to the GPU memory in the server, whereas time *"From Server"* refers to the movement of output data in the opposite direction. On the other hand, Figure 5.2(d) depicts a percentage comparison of the times shown in Figures 5.2(a), 5.2(b), and 5.2(c). Curves labeled with the *"Comm"* suffix in Figure 5.2(d) refer to the total communication time (comprising both *"To Server"* and *"From Server"* times). In general, it can be derived from the four figures that as input data transfer time decreases because of a faster network, the percentage of time devoted to computation increases. Figure 5.2(d) shows that the computation time is about 10% in the case for QDR, and it increases to nearly 20% when the InfiniBand EDR network adapter is used. Notice also that the percentage of the plan creation decreases when transfer size increases, since the time for creating a plan in the remote server requires an almost constant time of 340 ms.

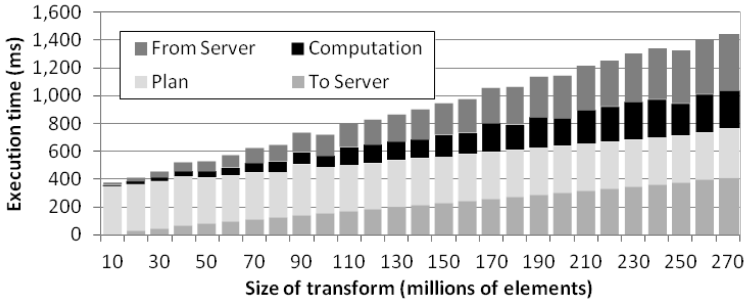
When multi-dimensional DFT transforms are considered, noticeable reductions in execution time are also achieved. Figure 5.3 shows the estimated execution time and attained speed up for 2D and 3D DFT transforms when using our middleware along with an improved communication layer. The same scenarios considered in Figure 5.1 for the 1D transform are also used in Figure 5.3. It can be seen that for the largest transform sizes, an speed up between 10x and 20x is achieved both for the 2D and 3D transforms. In the case of the 2D transform, even larger speed ups are attained for some transform sizes. Notice that, in general, speed up attained for multi-dimensional DFT transforms is slightly smaller than that achieved for the 1D FFTW function. One of the reasons, in addition to the different computation per element ratio shown in Figure 4.6, is the very different time required for creating the plan. In this regard, the time for 1D FFTW plan



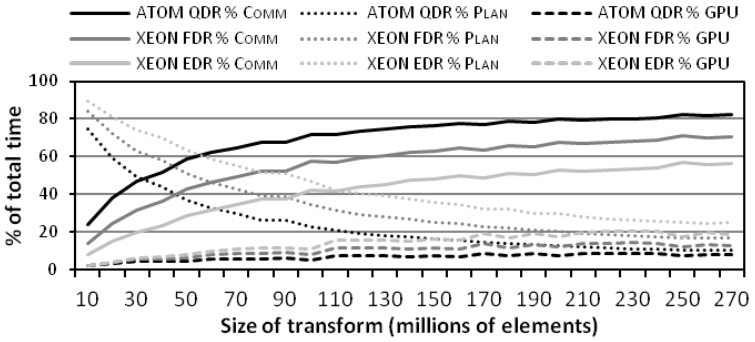
(a) Atom system using QDR network adapter.



(b) Xeon system with FDR network adapter.



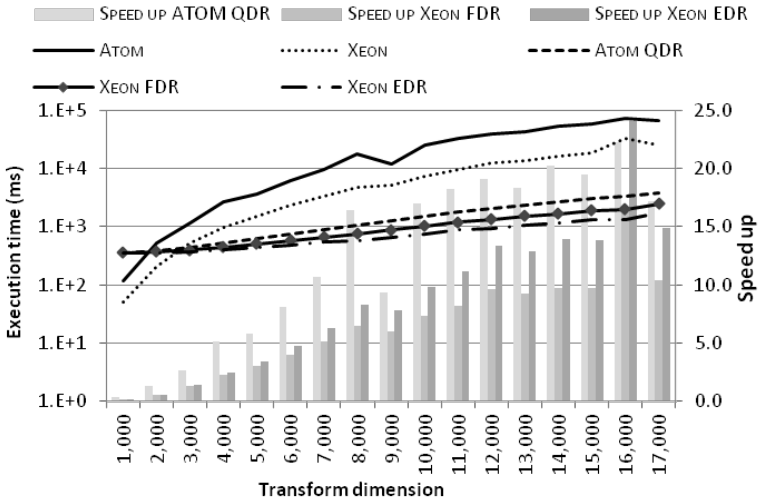
(c) Xeon system with EDR network adapter.



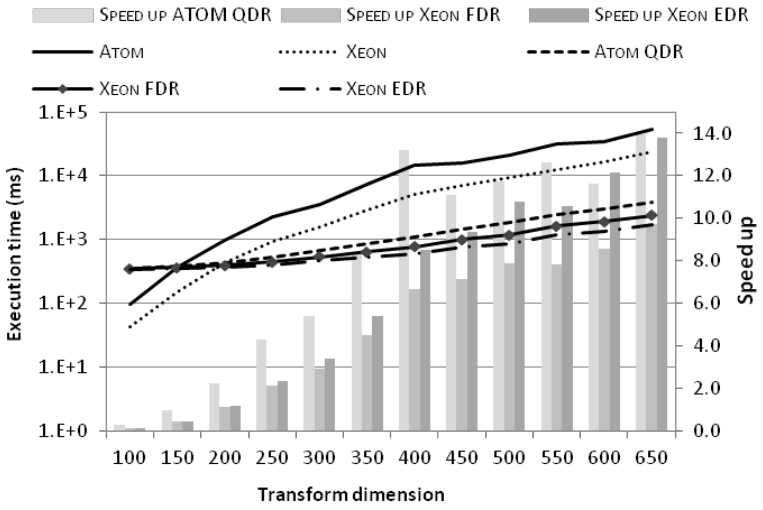
(d) Comparison among the three systems.

FIGURE 5.2: Execution time breakdown for the FFTW 1D function.

creation in the local Xeon 2.1 GHz ranges from 130 to 3485 ms depending on transform size, whereas creating the plan in the remote GPU server requires an almost constant



(a) FFTW 2D function.



(b) FFTW 3D function.

FIGURE 5.3: Performance estimation of the FFTW 2D and 3D functions.

amount of time around 340 ms. In this way, when outsourcing the 1D FFTW transform, the reduction in the time required for creating the plan causes an increment in the speed up. On the contrary, creating the plan for 2D and 3D transforms in the local Xeon 2.1 GHz requires just a few milliseconds[1], thus causing that speed up is reduced when the FFTW function is offloaded to the remote GPU, where creating the plan still requires about 340 ms.

[1]Time for plan creation mainly depends on the number of elements in each dimension. In this way, for a similar total transform size, 2D and 3D transforms present a much smaller number of elements per dimension than 1D DFTs.

Figure 5.3 also shows the effect already pointed out in previous chapter about the execution time degradation for the smallest transform sizes, given that local executions in the Xeon and in the Atom processors require less time than remote executions, thus suggesting that the new middleware should not blindly offload every computation but it should consider data size and network bandwidth in order to determine whether to remotely or locally execute the requested computation. This may be easily achieved with an automatic configuration stage to be performed just after installation of the new middleware in a given cluster.

Finally, remember that in Chapter 3 we have presented an optimization consisting in creating the plan for the DFT in the remote server in advance, before input data actually arrives. Thus, one may wonder whether this optimization, which will increase the programming complexity of the middleware as already discussed, reports significant benefits. In order to answer this question, we have overlapped the time of the plan creation with the time that input data takes to arrive to the GPU (labelled as *"To Server"* in Figures 5.2(a), 5.2(b), and 5.2(c)). Figure 5.4 presents the percentage of reduction in execution time among both versions when the three InfiniBand interconnects considered in the study are used along with the 1D transform. The time savings introduced by the optimization is in the range between 10% and 30%, mainly depending on transform size and network bandwidth. Actually, this dependence on the number of elements of the transform and on the performance of the network was the expected behavior, given that the proposed optimization uses input data transfer time (which depends on the two parameters mentioned) to hide the time required for creating the plan, which is almost constant when the GPU is involved. Notice that similar results were obtained for the multidimensional transforms.

5.2 Improved OpenBLAS

Chapter 3 has described the coarse idea for offloading the functions within the OpenBLAS library along with an optimization for the DGEMM function. In this chapter we present an estimation of the execution time as we did in the previous section with the FFTW library. The optimization mentioned in the previous chapter for the DGEMM function has not been considered in the first plots of this section.

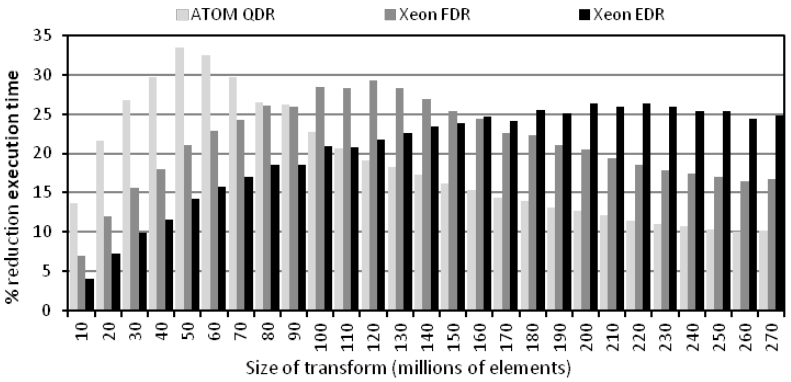


FIGURE 5.4: Reduction in the estimated execution time of the FFTW 1D function due to overlapping plan creation with input data transfer.

also considering the mentioned optimization for the DGEMM function.

Figure 5.5 presents the estimated performance for the DGEMM function. The label *"Xeon Local"* refers to the execution of the DGEMM function in the local Xeon 2.1 Ghz. The labels *"Xeon FDR"* and *"Xeon EDR"* refer to the system configurations where the network adapter was either a Mellanox InfiniBand FDR or EDR.

Figure 5.5 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. Figures 5.6(a) and 5.6(b) present the breakdown of the total estimated execution time when using the new middleware with the FDR and EDR network adapters, respectively. Execution time is split into three components: (1) time required to move the data from main memory in the client node to the GPU memory

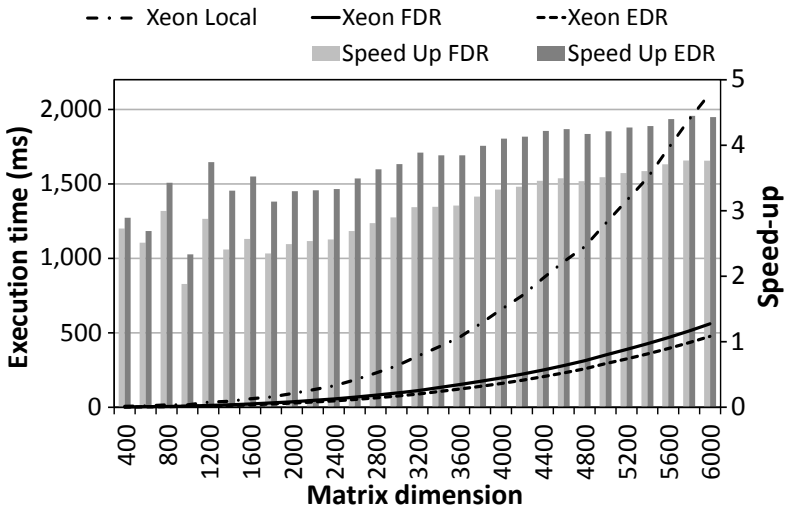
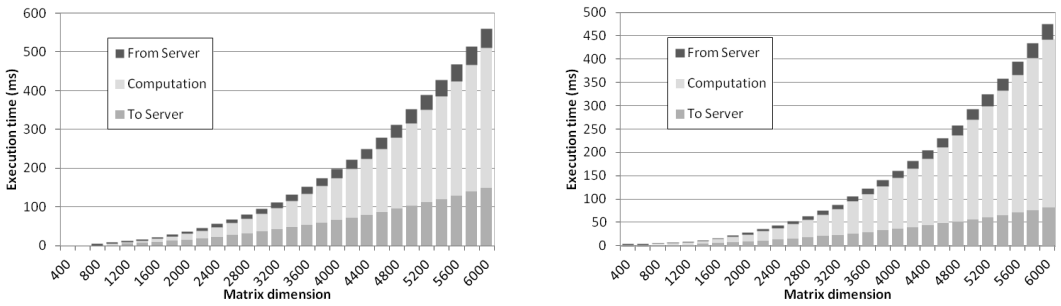


FIGURE 5.5: Estimated execution time for the DGEMM function.



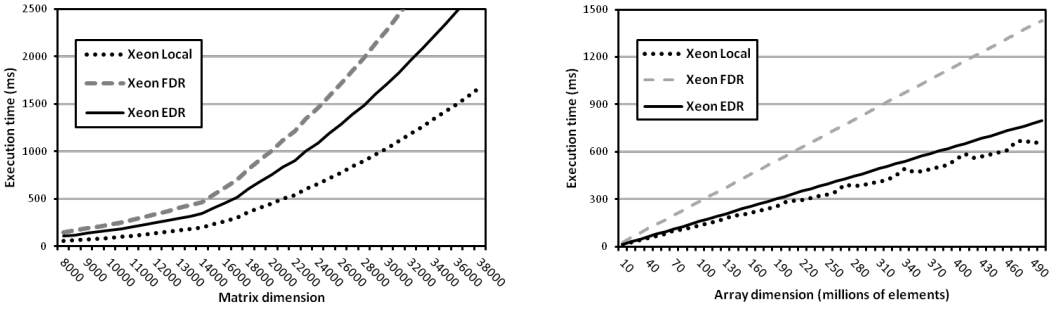
(a) InfiniBand FDR. (b) InfiniBand EDR.

FIGURE 5.6: DGEMM breakdown of estimated time execution.

(*"To Server"*), (2) time required to make the actual execution of the DGEMM function in the GPU (*"Computation"*), and (3) time required to move the data back to the client from the GPU memory (*"From Server"*). It can be seen 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.

On the other hand, 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 (Figure 5.7), no performance gain is attained for these functions.

Finally, Figure 5.8 shows the estimated performance when the optimization discussed in Chapter 3 is used. It can be seen that splitting the overall DGEMM computations into



(a) DGEMV execution time. (b) DDOT execution time.

FIGURE 5.7: Estimated execution time for DGEMV and DDOT.

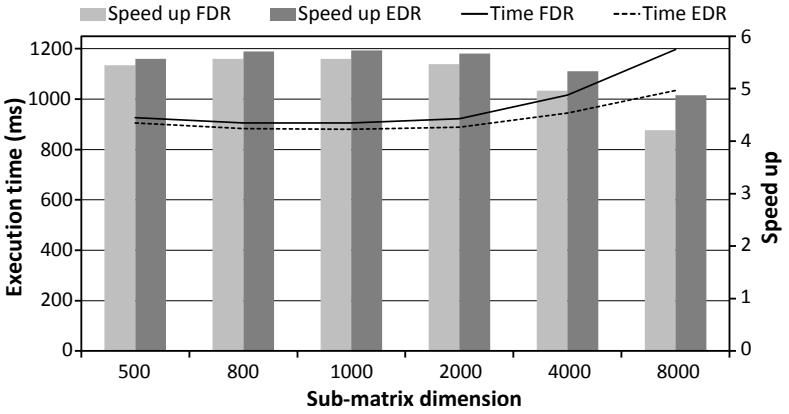


FIGURE 5.8: Performance estimation when the DGEMM operation in the remote GPU is pipelined. Square 8000x8000 matrices are used.

smaller operations reduces total execution time by 20% for the case of using 8 blocks per input matrix. In summary, Figure 5.8 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 distribution is an open research issue to be analyzed.

Chapter 6

Discussion

The new middleware presented in this thesis requires some additional discussion. First of all, we carried out an initial contact with the LAPACK library [29], since it is one of our future works. LAPACK is a software package intended to solve several mathematical functions related with linear algebra, such as linear least square problems, systems of linear equations and eigenvalue problems, among others. Functions within LAPACK can be classified into three different categories: driver routines, computational routines, and auxiliary routines. Figure 6.1 shows a traditional flow of a driver routine, which typically uses a sequence of computational routines in order to solve the problem. Auxiliary routines are intended to perform certain subtasks and common low-level computations and are not usually employed by LAPACK users but by driver and computational routines.

The behavior of driver routines perfectly matches the idea of the new middleware. In this regard, once a driver routine is called by the application, the associated wrapper would forward to the remote server the input data to that driver routine. At the server, the driver routine would be executed in the GPU, involving several computational routines. Once all the computations have been performed and the execution of

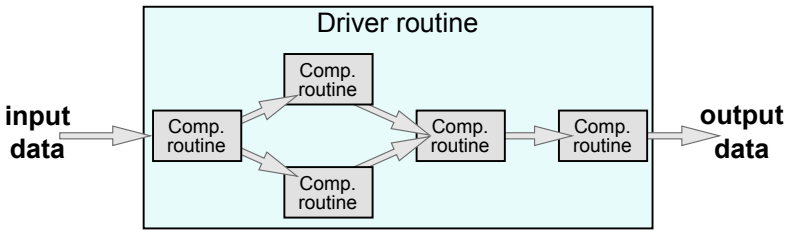


FIGURE 6.1: Relationship between driver and computational routines in LAPACK.

the driver routine has been completed at the GPU, results would be returned back to the client side of the middleware. As can be seen, driver routines present a very high computation-to-communication ratio, what will make them very good candidates for being remotely accelerated, given that the communication overheads will be amortized by the larger amount of computations performed. Thus, the main concern in the proposed middleware, which is the overhead of communications, should be minimized for driver routines. Regarding computational routines, these ones could also be offloaded to the remote accelerator, in the same way as functions from BLAS and FFTW were outsourced.

Considering these issues, we have estimated the performance of the proposed middleware when it is applied to the SGESV function and the optimized communications explained in the previous chapter are set. This function computes the solution to a real system of linear equations $B = A * X$ where A is an n-by-n matrix and X and B are n-by-nrhs matrices (nhrs stands for number of right hand sides, that is, the number of columns of matrix B). The SGESV function internally makes use of the LU decomposition with partial pivoting and row interchanges in order to factor matrix A. Once matrix A is factored, the result is used to solve the system of equations $B = A * X$. As can be seen, this driver routine does not make an intensive use of computational routines. Figure 6.2 shows a comparison between the execution time of this function using all the 12 available cores in the system and the estimated execution time attained by the new middleware. The MKL library has been used for the CPU-based executions to make a fair comparison. In the case of the new middleware, the MAGMA library has been selected to carry out

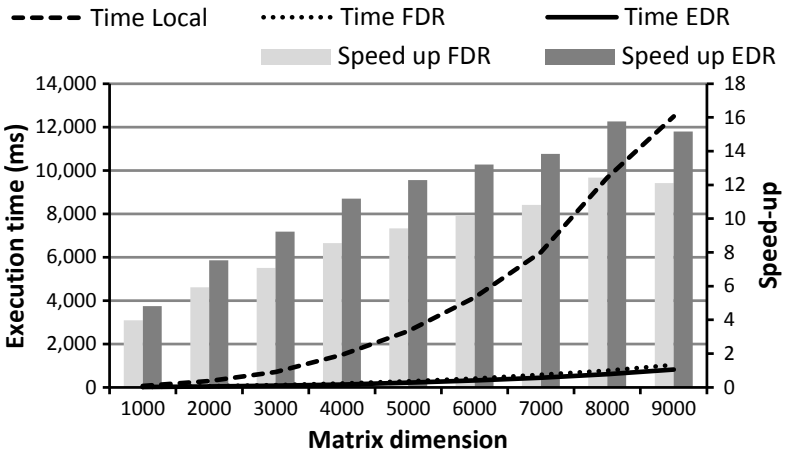


FIGURE 6.2: Performance estimation of the SGESV function using InfiniBand FDR and EDR networks.

the computation of this function in the GPU. It can be seen in the figure that the use of the new middleware would provide noticeable improvements in execution time for both the FDR and EDR InfiniBand networks with up to 16x speedups. Notice also that MAGMA is able to use several GPUs for many of the functions it addresses. Thus, when several GPUs are used, speedups shown in Figure 6.2 might be larger.

But there are also some additional concerns regarding the libraries studied in this work. First, the evaluated problem sizes range from 10 to 270 million of elements for 1D DFTs. However, how should larger transform sizes be managed? Notice that available memory at the GPU is the limiting factor. Hence, given that the Tesla K40 features 12 GB of GDDR5, then transform sizes up to 400 million of elements could be considered. For larger DFT sizes, the DFT operation can be divided into smaller DFTs [30] which can be independently computed and later combined. Therefore, if problem size exceeds GPU memory, the new middleware should automatically split input data into smaller chunks that individually fit into GPU memory so that results are later combined before being delivered to the application executing the original `fftw_execute` function. This splitting and combining process should be transparent to the application. In the case of BLAS operations, a similar decomposition should be considered.

One more concern to be discussed regarding the FFTW library is that many use cases of 3D DFT in scientific computing involve domain decomposition with slab or pencil type subdomains, coupled with an alltoall type transpose communication. This enables the use of 1D DFTs to solve 3D DFT problems. Additionally, domain decomposition allows much larger problems to be handled with the GPU's limited on-board memory. Thus, in a similar way to the decomposition of 1D DFTs mentioned above, multi-dimensional DFTs should also be automatically managed by the new middleware in a transparent way to the application, which just performed a call to the FFTW library to carry out a multi-dimensional DFT. On the contrary, in case this decomposition is already managed by the application, then the new middleware would manage 1D DFT transforms and should only address whether the 1D transform requested fits into GPU memory, as explained before. Although this concern has been explained for the FFT library, it applies in the same way for the BLAS and LAPACK libraries.

Finally, notice that in addition to the GPU, the remote server also features one or more CPU sockets, which have not been used in this implementation of the new middleware in

order to further accelerate the execution of the functions. However, the new middleware is not tied to the use of the NVIDIA cuFFT and cuBLAS libraries in the remote server, but other libraries could also be used. In this regard, more complex libraries that distribute the computations among the GPU and the available CPU cores could also be used if available. In a similar way, libraries that leverage several GPUs could also be used in those cases when the remote server features more than one GPU.

Chapter 7

Related Work

In this chapter the new middleware is put into the right context. In this regard, offloading computations to specialized hardware or servers has been previously proposed in a variety of contexts, such as heterogeneous multicores within a chip, GPU offloading within a computing node (this is what NVIDIA does with CUDA), and scheduling the entire job to specialized nodes in grid-computing infrastructure. Their difference with the new middleware is that the latter works at the cluster level instead of at the node domain.

At the cluster level, the new middleware may resemble the remote procedure call (RPC) mechanism [31] [32], which is a well-established technique to allow function shipping to remote nodes using function skeletons. However, there are important differences between the RPC mechanism and the new middleware. First, in order to use RPCs, programmers have to explicitly program their use whereas the new middleware does not require any modification to application source code. Second, contrary to the RPC mechanism, the new middleware does not require the intervention of the operating system to move data to the remote server (for instance, in the optimized communication layer using the InfiniBand Verbs API is used, all the code is executed at the user level without requiring a context switch). Furthermore, the new middleware does not use client or server stubs neither in the caller nor in the callee machine, as RPCs do. Moreover, an RPC runtime is neither required: the new middleware is integrated within the application at the client side and, at the server side, the server directly deals with computation requests, without requiring the intervention of any other entity. Also, the new middleware does not require

the use of specialized compilers such as `rpcgen` or the use of functions from the RPC Programmer's Interface such as `rpc_reg()` or `rpc_call()`. Finally, the RPC mechanism is stateless (each RPC call is independent from each other) whereas the new middleware requires to store several state variables between different calls from the application, such as the fake plans described in Chapter 3.

Other previous proposals that may also resemble the new middleware are Object Request Brokers (ORB) [33], which consist of a broker process running in a network that puts into contact a client requesting a service with a server providing that service. In the ORB approach, the programmer either uses an Interface Definition Language (IDL) to declare the public interfaces of the server or the compiler of the used programming language translates the language statements into IDL statements. In contrast, the new middleware does not require a centralized entity to put into contact requesters with servers nor it requires to modify the application code nor specialized compilers. Moreover, on the ORB's client side, stub objects are created and invoked, serving as the only visible part within the client application. Contrariwise, the new middleware does not require any kind of stubs.

In summary, the main differences among other proposals and the new middleware are (1) the latter does not require any modification to the source code of applications (it does not even require applications to be recompiled as long as they use dynamically linked libraries) and (2) the new middleware does not require stubs or additional runtime systems to work or specialized compilers.

Finally, notice that this middleware can make use of GPUs in the remote server, as remote GPU virtualization frameworks do. However, there is an important difference between both: remote GPU virtualization frameworks redirect GPU requests from their original destination (a local GPU) to a remote GPU, whereas the new middleware moves the computation of compute-intensive libraries, initially intended to be performed at the local CPU cores, to a remote accelerator thus transforming the nature of the call in order to execute a different instantiation of the code, which provides exactly the same results. Furthermore, notice that outsourcing the computations to be performed within scientific libraries with the new middleware reports execution time savings. On the contrary, remote GPU virtualization frameworks do not accelerate the computations

forwarded to the remote server but they are usually slightly penalized due to the longer path to the remote GPU with respect to the case of using it locally.

Chapter 8

Conclusions

In this thesis we have presented the initial implementation of a new middleware that offloads the CPU-based computations of scientific libraries to remote accelerators located in other nodes of the cluster. Moreover, we have conducted a thorough performance evaluation of the new middleware, which partially supports the FFTW and OpenBLAS libraries as well as CUDA-compatible GPUs. Results clearly show that this new framework provides important reductions in execution time. Results also show that performance greatly depends on the throughput of the underlying network fabric.

Although the current version of the new middleware partially supports the FFT and BLAS libraries, future versions will fully support both and also the LAPACK library, as well as the use of Intel Xeon Phi accelerators and an optimized communications layer. Once the main scientific libraries are supported, the benefits that the new middleware may report to real applications, in terms of execution time reductions, should be evaluated. In this regard, it is important to remark that although the use of GPUs is being considered for many applications, there are other applications that, although being initially ported to CUDA, they later discontinued this support. This is the case, for example, for the DL_POLY application [34]. Other applications never were ported to GPUs, for instance the TINKER molecular modeling software package [35]. Therefore, for those applications not being ported to GPUs, the new middleware might be especially appealing in order to reduce execution time. Actually, the use of the new middleware with those applications may delay the need for adapting them to the use of GPUs or other accelerators. Notice also that the new middleware is compatible with

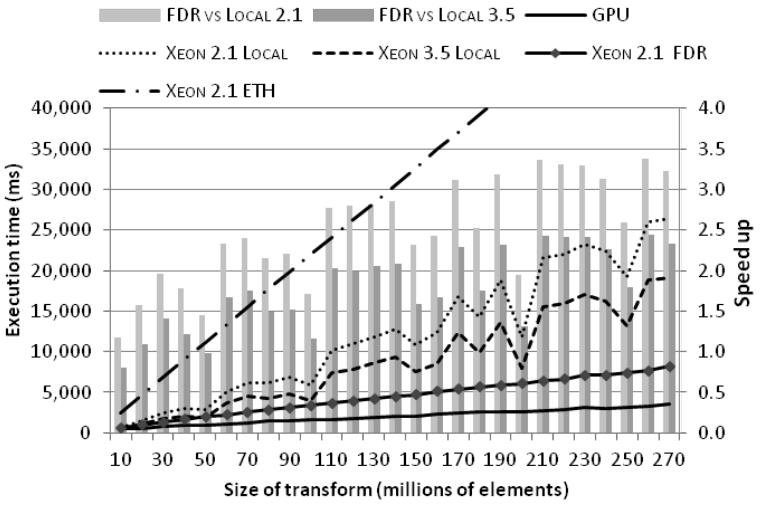
the use of MPI. In this case, each of the MPI processes would independently offload its computations to the remote server. Automatic load balancing among servers would be an open research topic.

Finally, although we have presented the use of the new middleware for the FFT and BLAS libraries, our main goal is targeting the complete Intel MKL library in the near future, so that all of its functions can be offloaded to remote accelerators. Other widely used scientific libraries intended to be executed on CPUs will also be considered in future versions. Additionally, concurrent usage of remote servers among several clients and scheduling issues are other open future research terms.

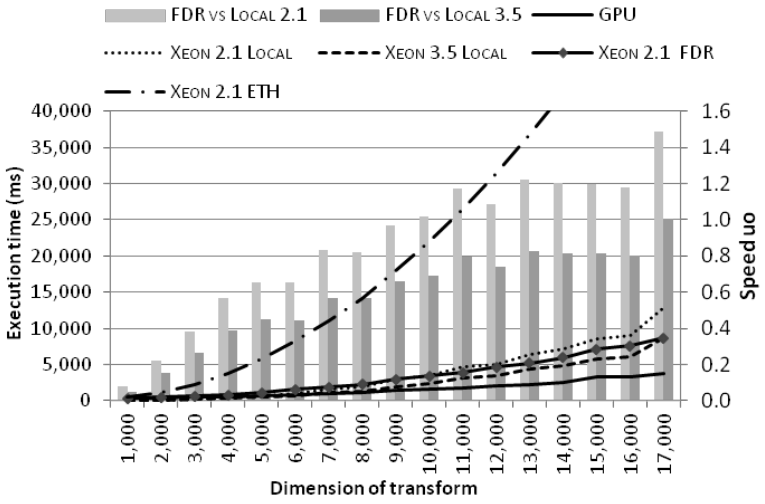
Appendix A

Performance Evaluation for C2R and R2C DFT

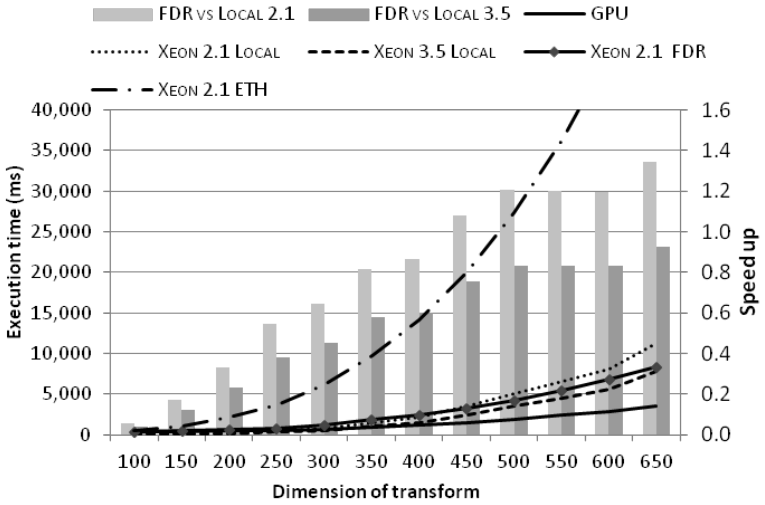
As a complement of the results depicted in Chapter 4, we present in this appendix the performance evaluation of the FFTW functions involving real data in the input (real-to-complex transform, R2C) or in the output (complex-to-real transform, C2R).



(a) 1D.

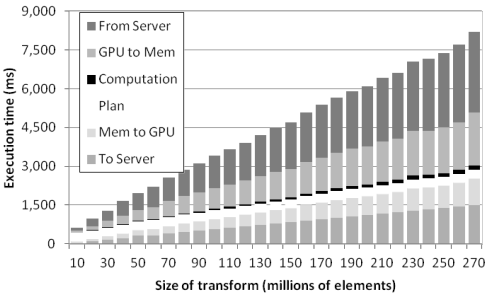


(b) 2D.

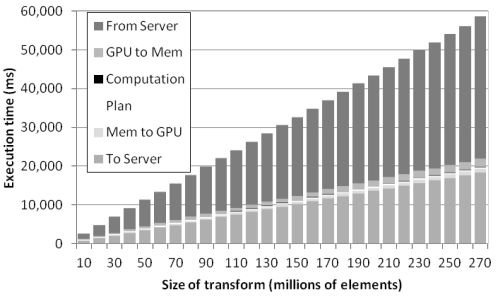


(c) 3D.

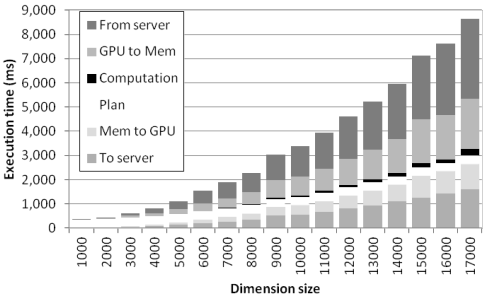
Figure A.1: Performance of the R2C-DFT function in the Xeon-based system.



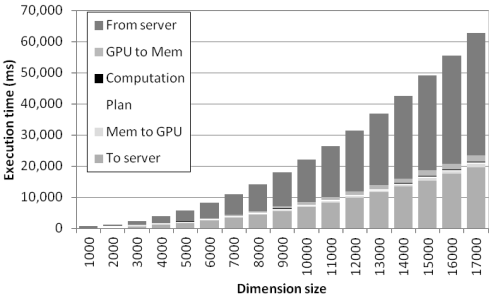
(a) 1D, InfiniBand FDR.



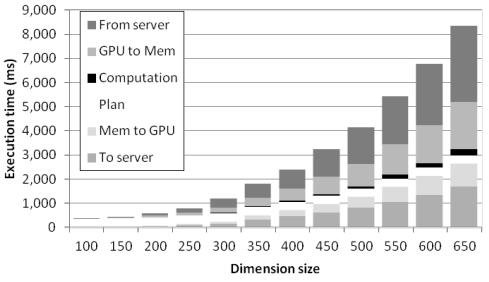
(b) 1D, Ethernet.



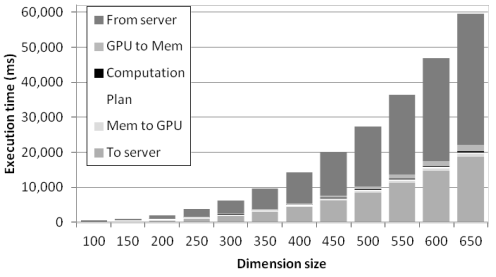
(c) 2D, InfiniBand FDR.



(d) 2D, Ethernet.

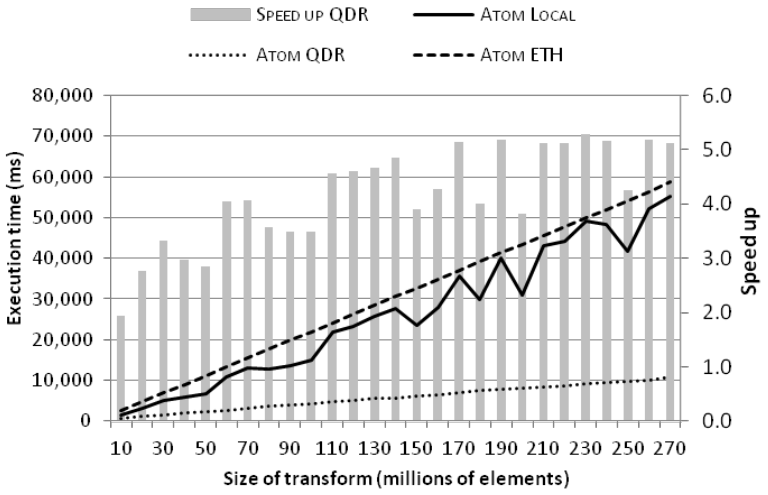


(e) 3D, InfiniBand FDR.

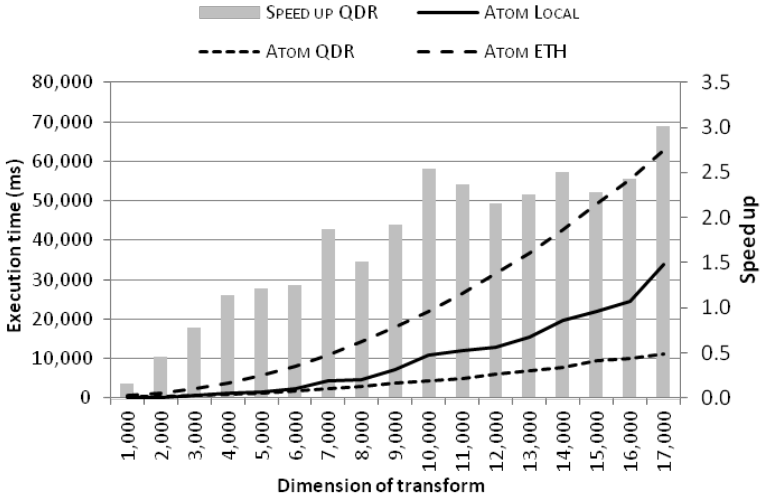


(f) 3D, Ethernet.

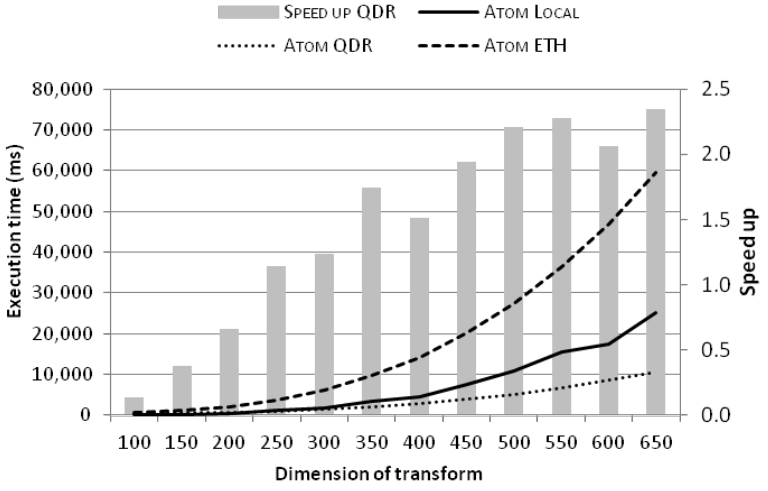
FIGURE A.2: Breakdown of R2C-DFT in the Xeon-based system.



(a) 1D.

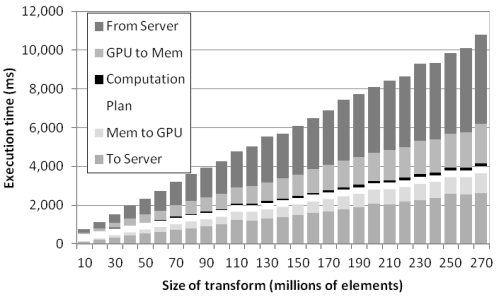


(b) 2D.

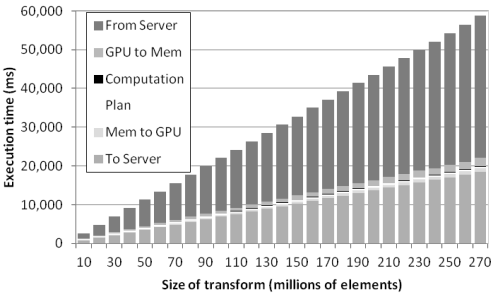


(c) 3D.

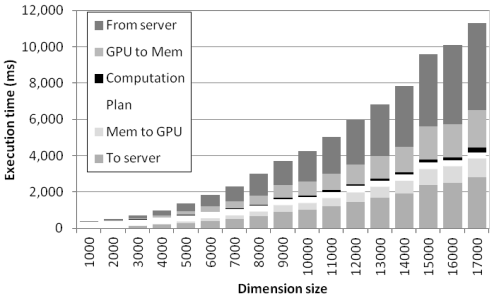
Figure A.3: Performance of the R2C-DFT function in the Atom-based system.



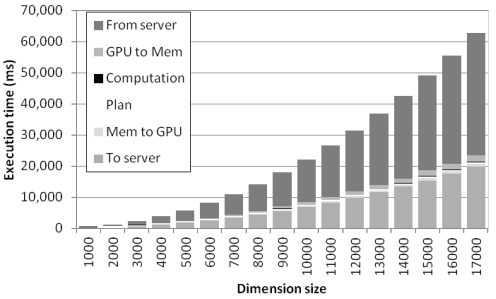
(a) 1D, InfiniBand QDR.



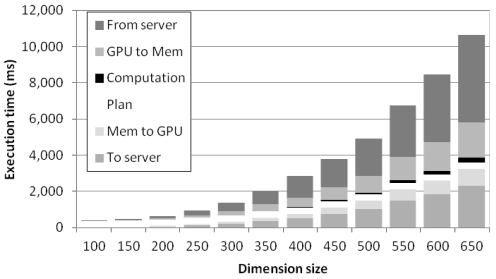
(b) 1D, Ethernet.



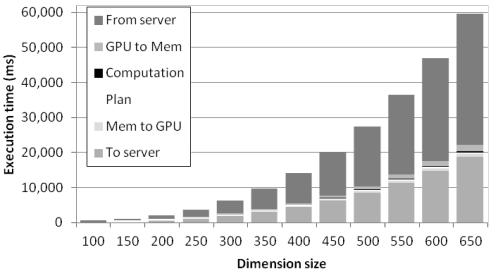
(c) 2D, InfiniBand QDR.



(d) 2D, Ethernet.

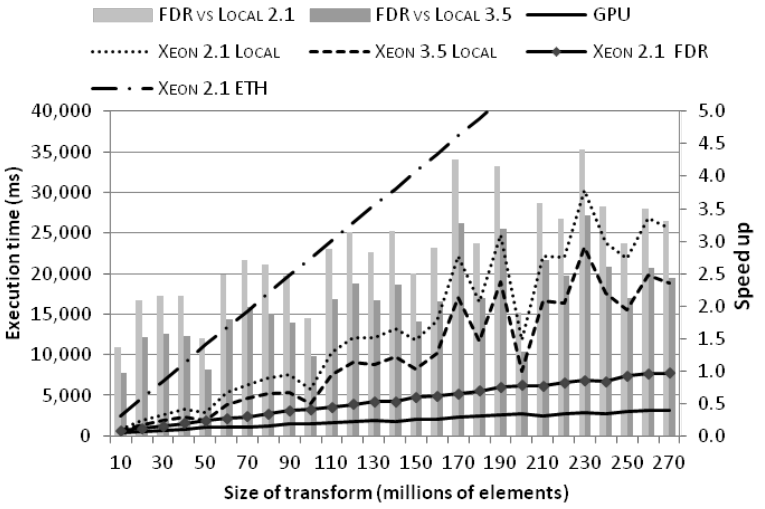


(e) 3D, InfiniBand QDR.

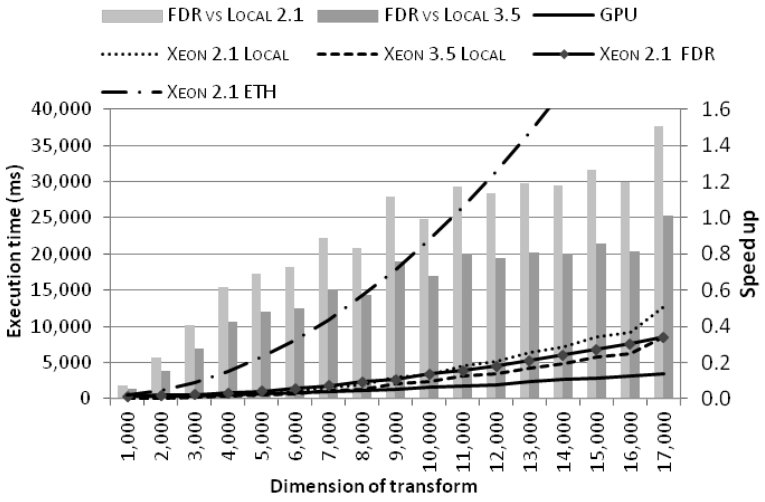


(f) 3D, Ethernet.

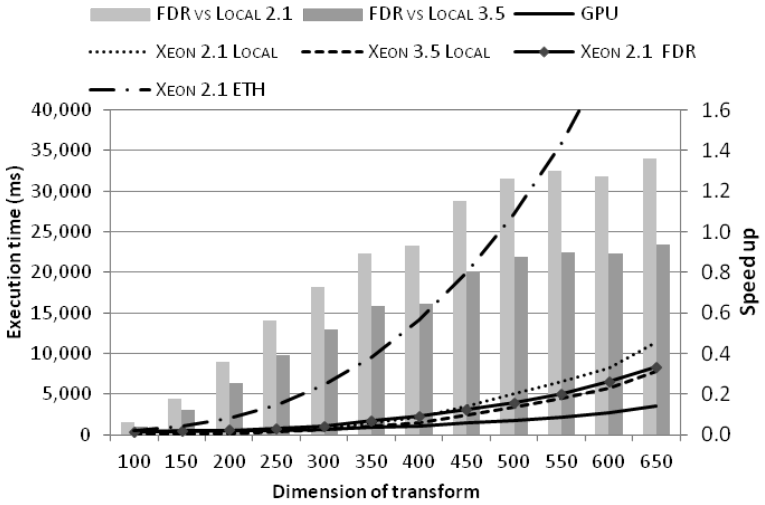
FIGURE A.4: Breakdown of R2C-DFT in the Atom-based system.



(a) 1D.

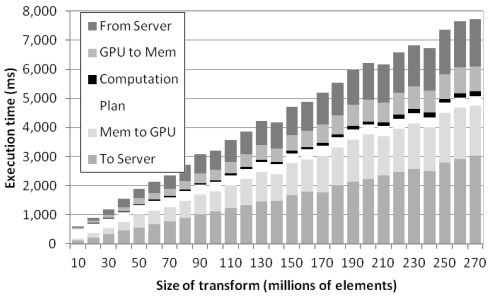


(b) 2D.

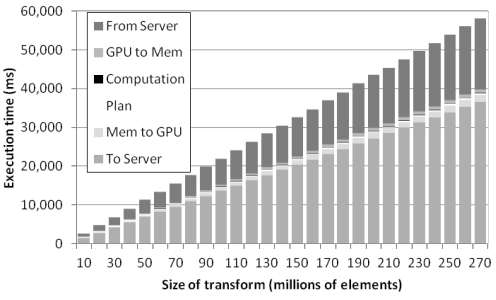


(c) 3D.

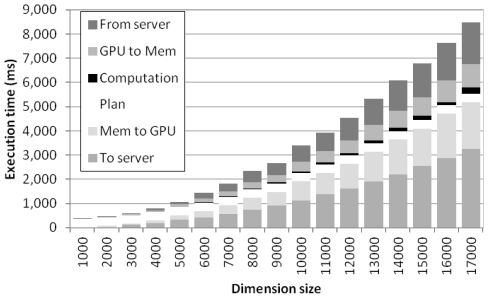
FIGURE A.5: Performance of the C2R-DFT function in the Xeon-based system.



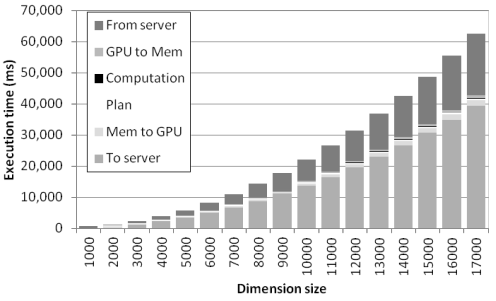
(a) 1D, InfiniBand FDR.



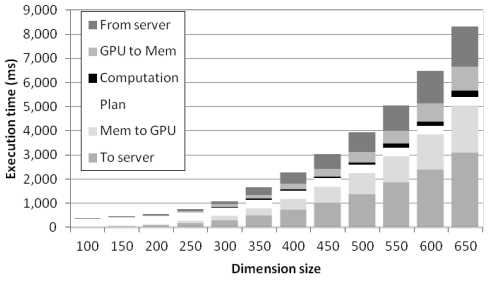
(b) 1D, Ethernet.



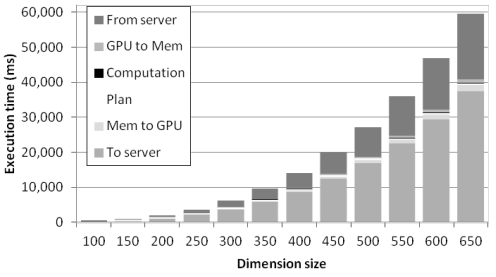
(c) 2D, InfiniBand FDR.



(d) 2D, Ethernet.

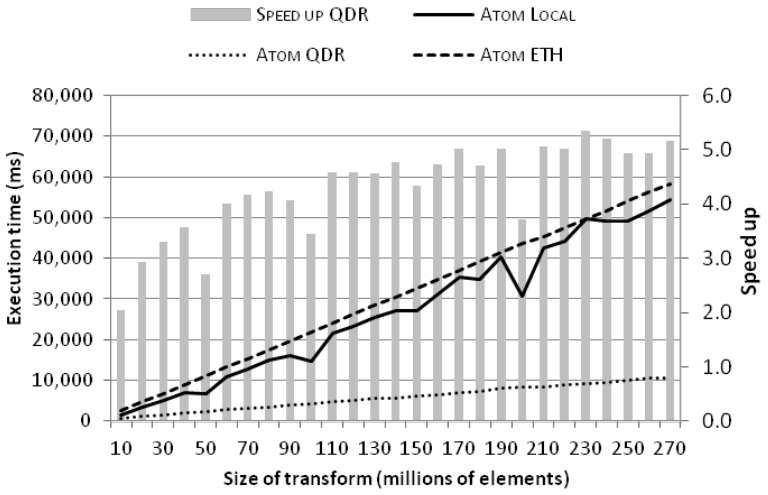


(e) 3D, InfiniBand FDR.

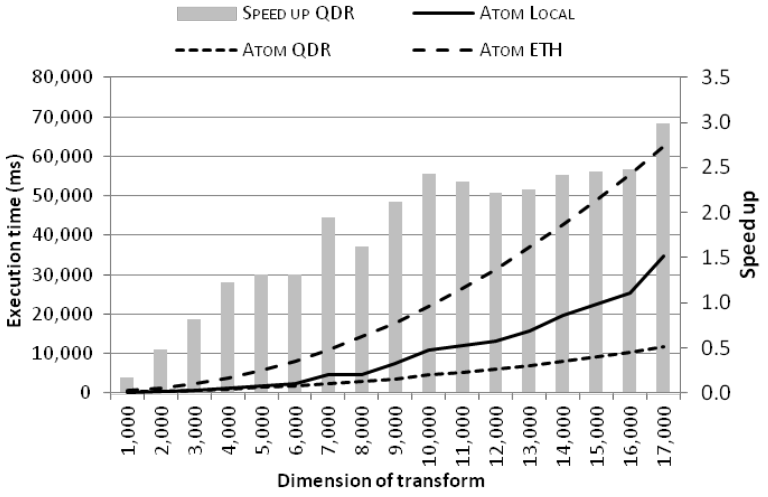


(f) 3D, Ethernet.

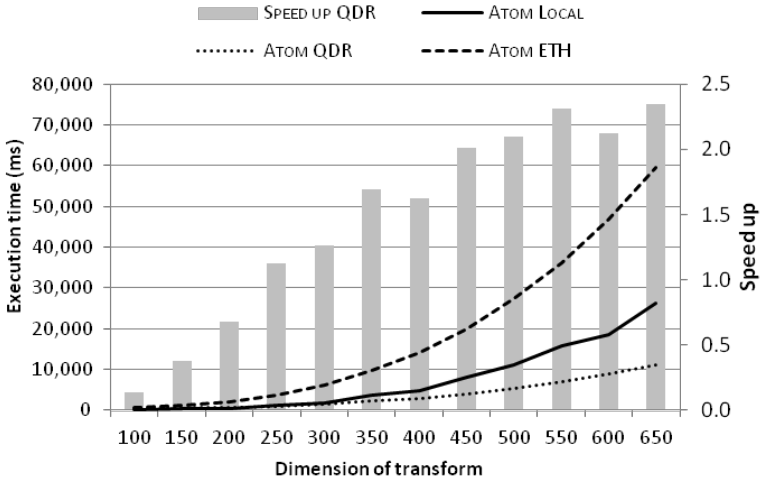
FIGURE A.6: Breakdown of C2R-DFT in the Xeon-based system.



(a) 1D.

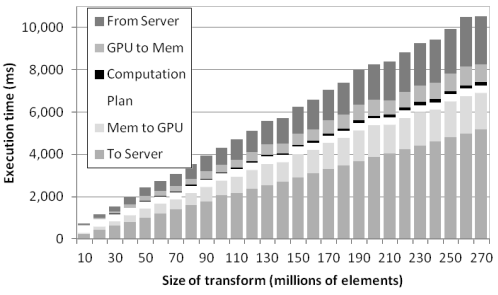


(b) 2D.

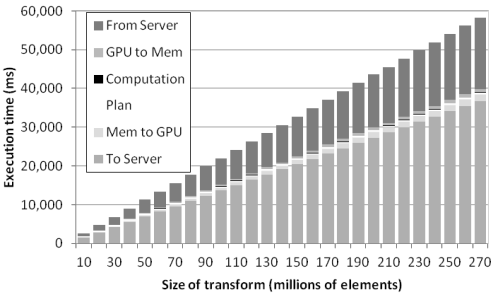


(c) 3D.

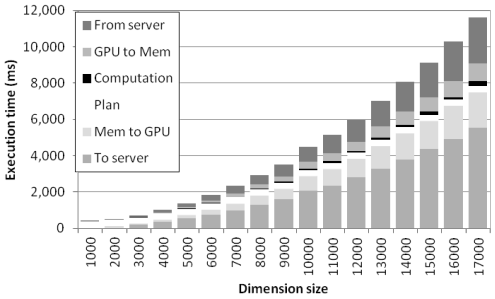
FIGURE A.7: Performance of the C2R-DFT function in the Atom-based system.



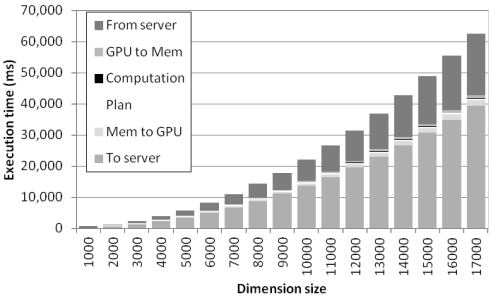
(a) 1D, InfiniBand QDR.



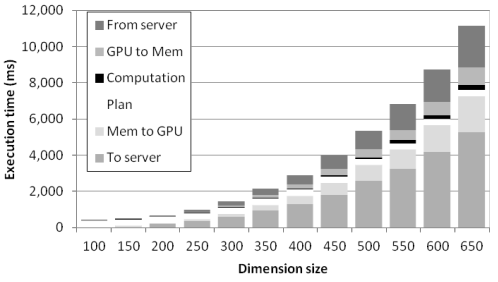
(b) 1D, Ethernet.



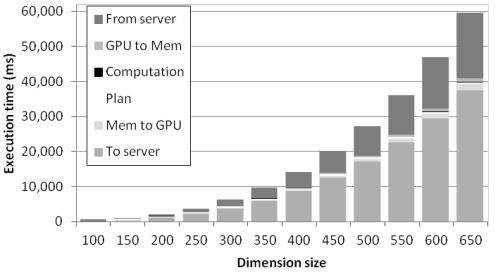
(c) 2D, InfiniBand QDR.



(d) 2D, Ethernet.



(e) 3D, InfiniBand QDR.



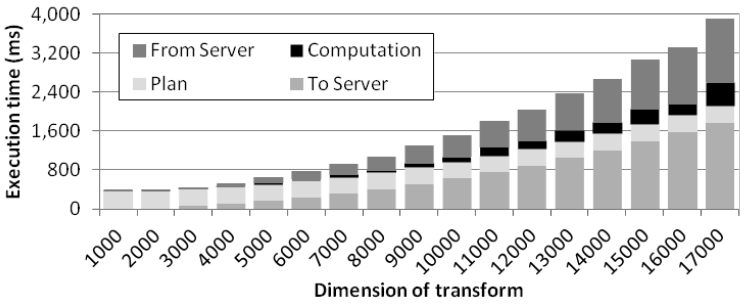
(f) 3D, Ethernet.

FIGURE A.8: Breakdown of C2R-DFT in the Atom-based system.

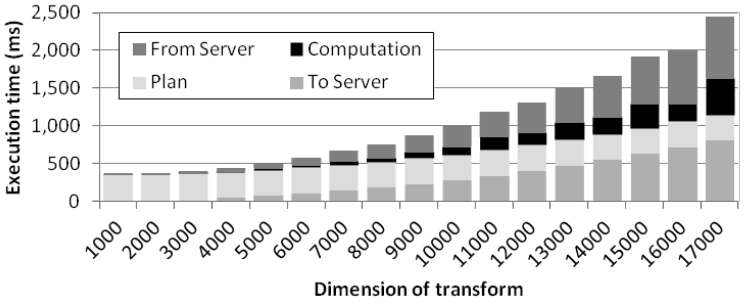
Appendix B

Performance Estimation for C2R and R2C DFT

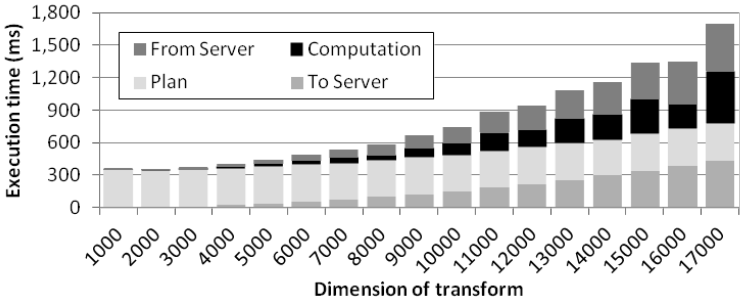
In this appendix we have included the execution time breakdown for the multidimensional DFT transforms with complex data in both input and output (complex-to-complex, C2C), not shown in Chapter 5. Furthermore, in the same way we did in Appendix A, as a complement of the results depicted in Chapter 5, we present in this appendix the performance estimation with improved communications of the FFTW functions involving real data in the input (real-to-complex transform, R2C) or in the output (complex-to-real transform, C2R).



(a) Atom system with QDR network adapter.

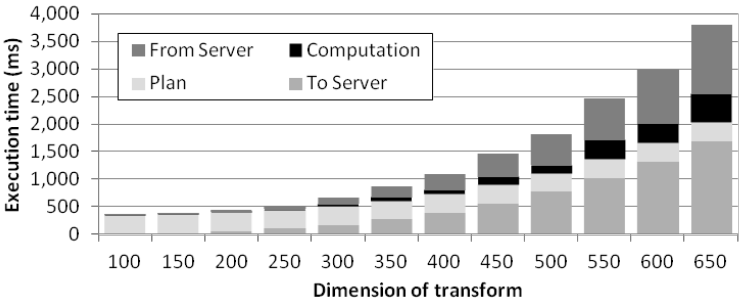


(b) Xeon system with FDR network adapter.

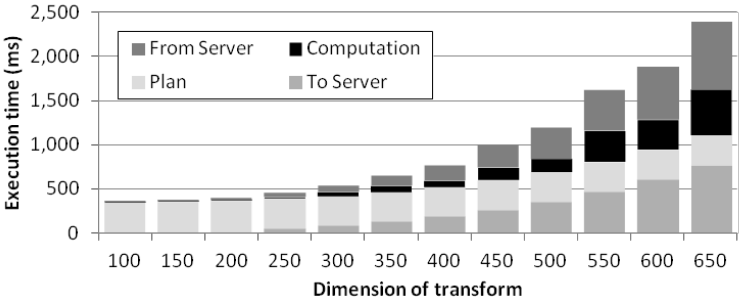


(c) Xeon system with EDR network adapter.

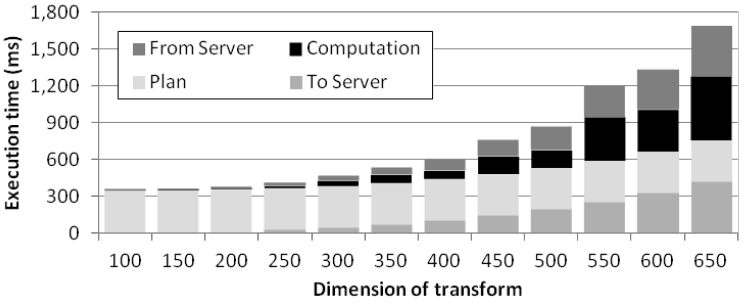
FIGURE B.1: Execution time breakdown for the C2C-DFT 2D.



(a) Atom system with QDR network adapter.

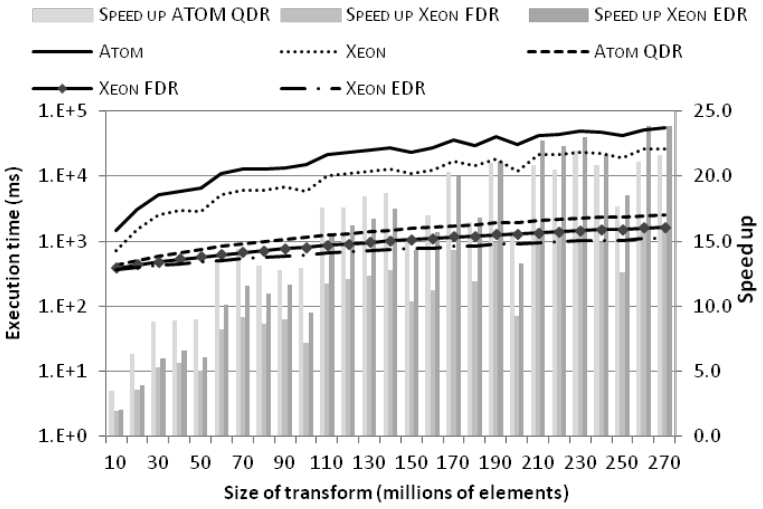


(b) Xeon system with FDR network adapter.

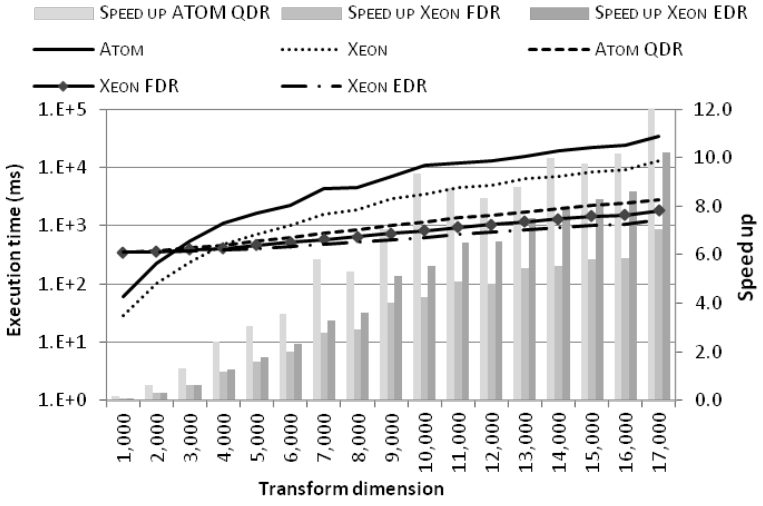


(c) Xeon system with EDR network adapter.

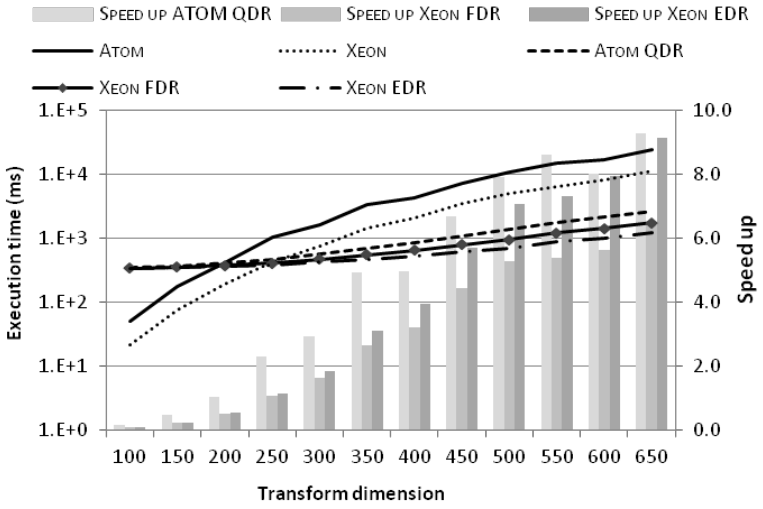
FIGURE B.2: Execution time breakdown for the C2C-DFT 3D.



(a) 1D.

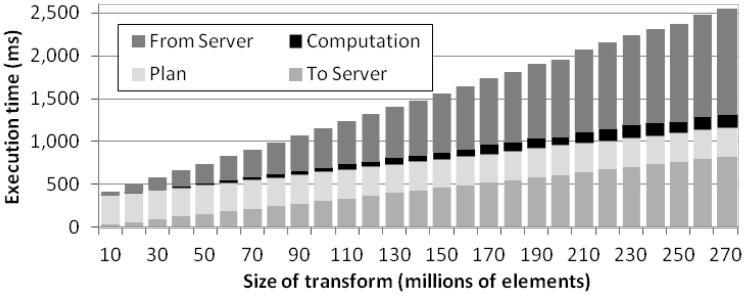


(b) 2D.

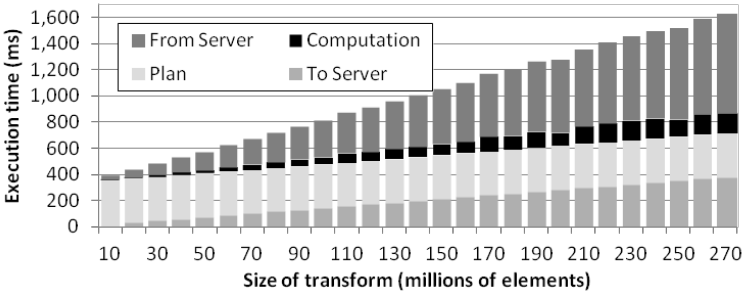


(c) 3D.

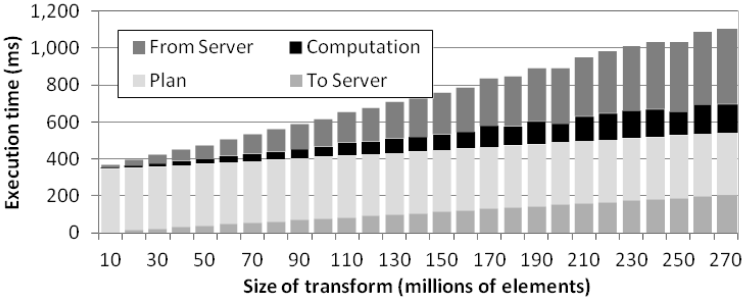
Figure B.3: Performance estimation of the R2C-DFT function.



(a) Atom system with QDR network adapter.

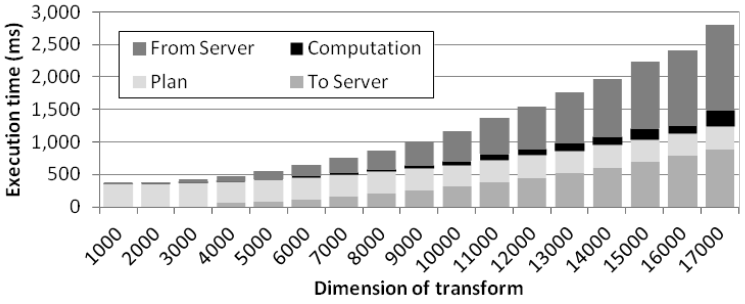


(b) Xeon system with FDR network adapter.

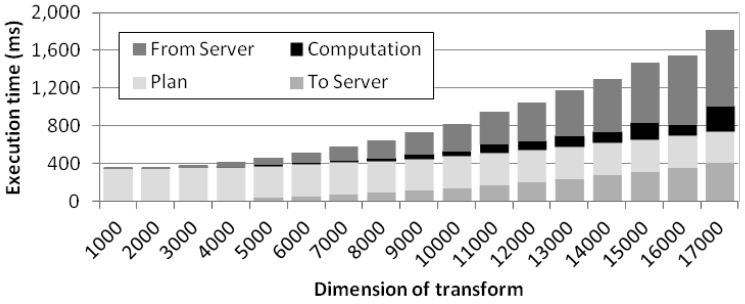


(c) Xeon system with EDR network adapter.

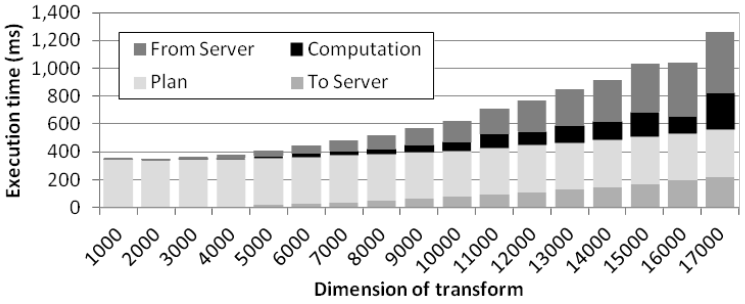
FIGURE B.4: Execution time breakdown for the R2C-DFT 1D.



(a) Atom system with QDR network adapter.

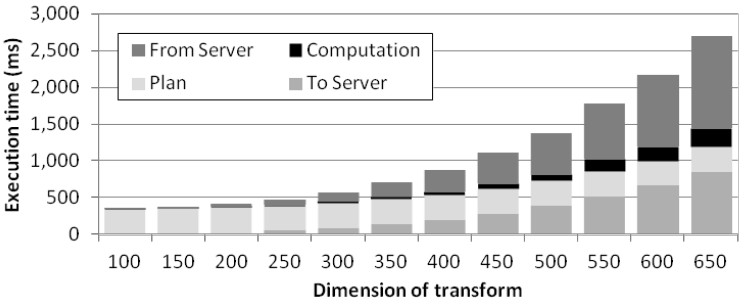


(b) Xeon system with FDR network adapter.

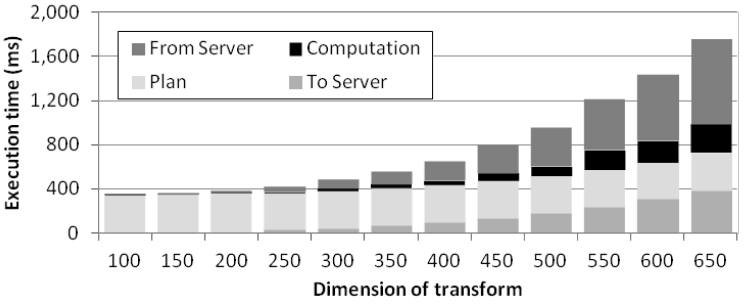


(c) Xeon system with EDR network adapter.

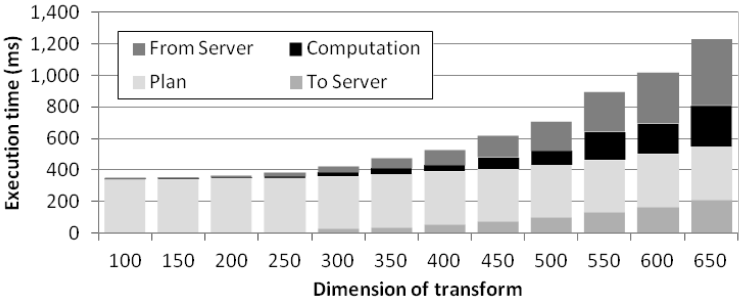
FIGURE B.5: Execution time breakdown for the R2C-DFT 2D.



(a) Atom system with QDR network adapter.

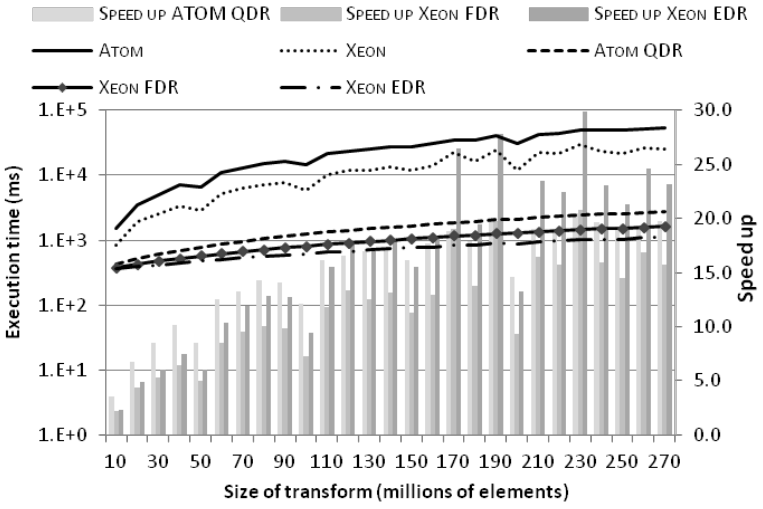


(b) Xeon system with FDR network adapter.

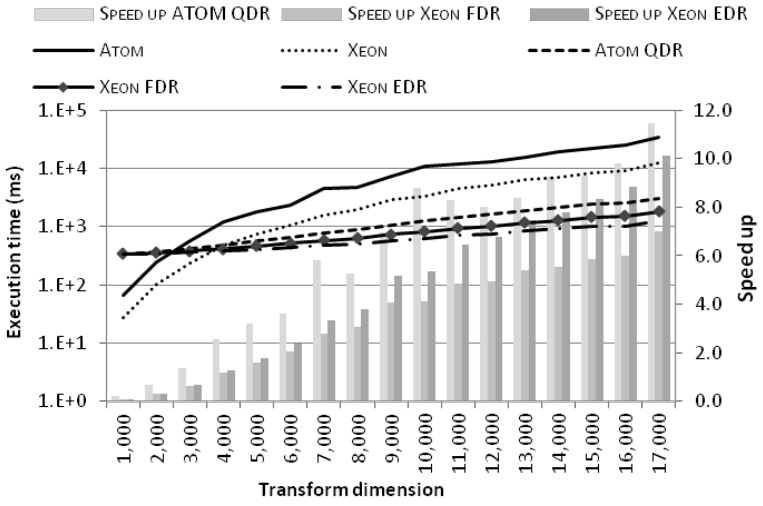


(c) Xeon system with EDR network adapter.

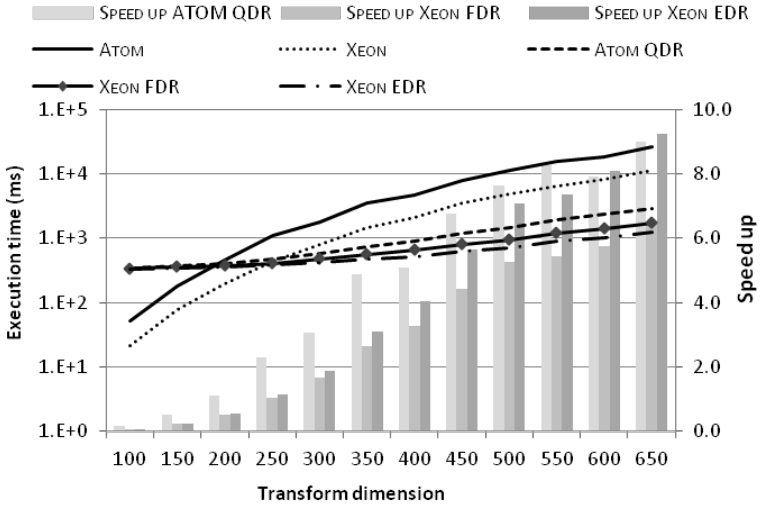
FIGURE B.6: Execution time breakdown for the R2C-DFT 3D.



(a) 1D.

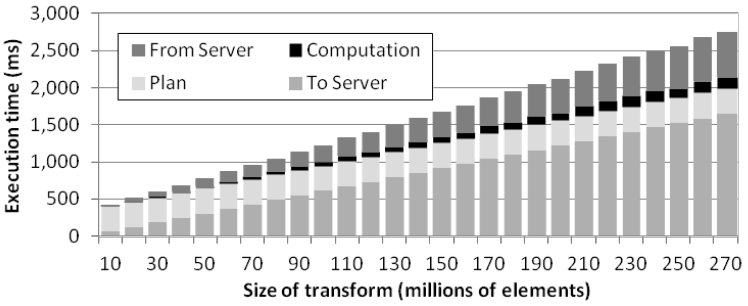


(b) 2D.

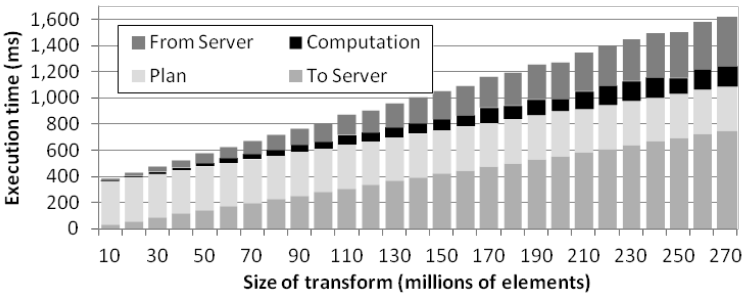


(c) 3D.

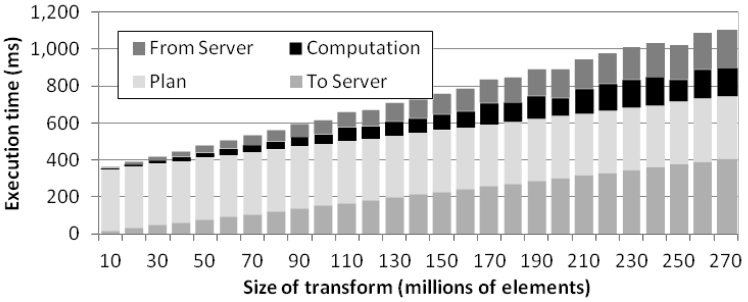
Figure B.7: Performance estimation of the C2R-DFT function.



(a) Atom system with QDR network adapter.

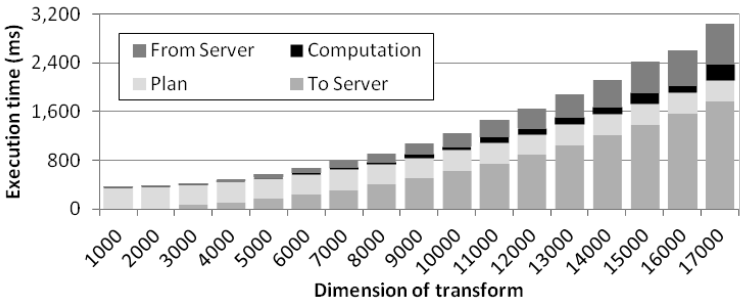


(b) Xeon system with FDR network adapter.

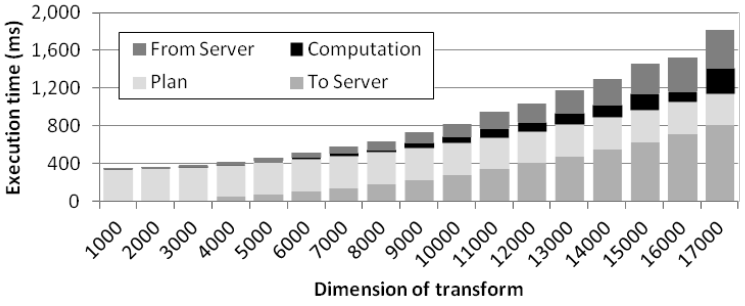


(c) Xeon system with EDR network adapter.

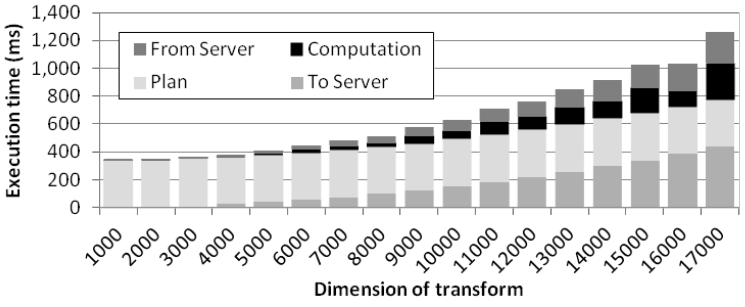
FIGURE B.8: Execution time breakdown for the C2R-DFT 1D.



(a) Atom system with QDR network adapter.

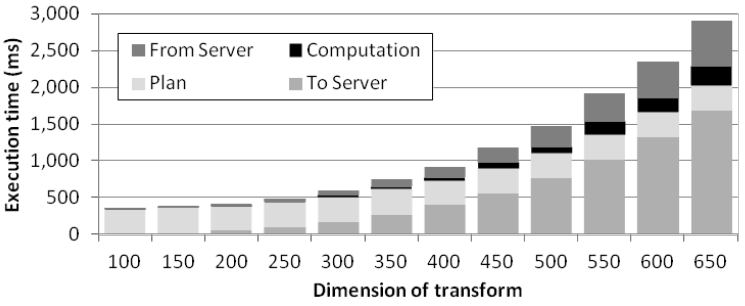


(b) Xeon system with FDR network adapter.

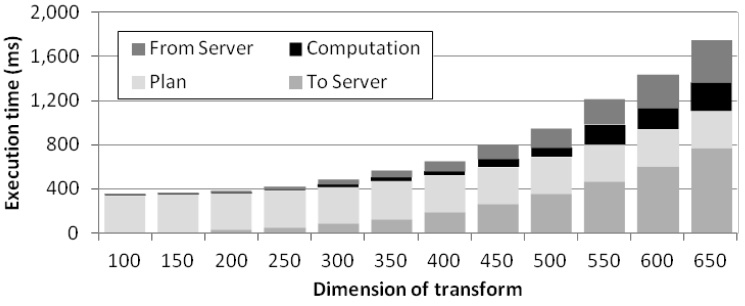


(c) Xeon system with EDR network adapter.

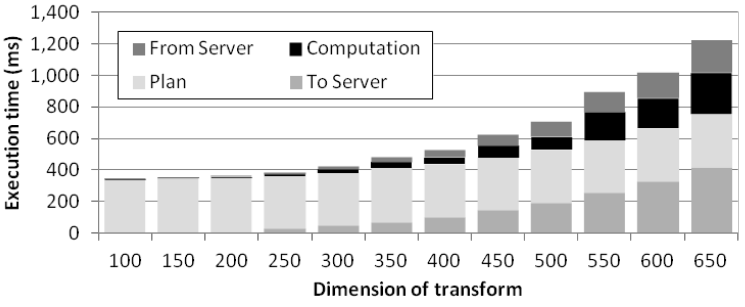
FIGURE B.9: Execution time breakdown for the C2R-DFT 2D.



(a) Atom system with QDR network adapter.



(b) Xeon system with FDR network adapter.



(c) Xeon system with EDR network adapter.

FIGURE B.10: Execution time breakdown for the C2R-DFT 3D.

Appendix C

Performance Evaluation of FFTW with ARM

In Chapter 4 we presented a performance evaluation of the middleware based on the results obtained with the the testbenches based on Xeon and Atom processors. A similar performance evaluation has also been carried out in an ARM-based system.

The ARM-based system is a Jetson TK1 DevKit [36] by NVIDIA comprising a 4-core ARM Cortex A15 CPU at 2.3 GHz and 2 GB of memory, version 21.4 of L4T (Linux for Tegra) and 1 Gbps Ethernet as a network fabric connecting it with the server. The lack of more memory and PCIe connectors in this system is the reason why the results obtained

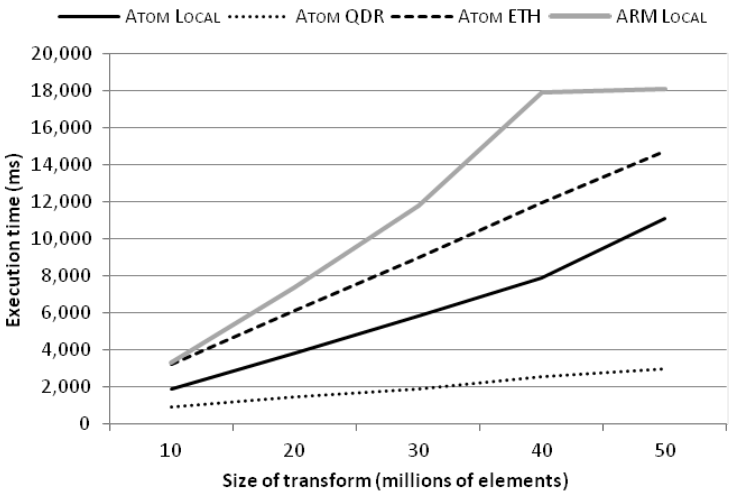


Figure C.1: Execution time of low-power based systems for the C2C 1D-DFT.

by the Jetson TK1 Devkit have not been included in the main study. Additionally, a problem with the 1 Gbps ETH driver in the new platform is that the bandwith of the Ethernet interface is downgraded to 100 Mbps.

With these limitations, and only having connectors of 1 Gbps Ethernet, the performance of this system with the FFTW library does not provide benefits. This result is aligned with the ones presented in Chapter 4 for both the Xeon and Atom-based systems using the same connector.

Appendix D

Performance Estimation of FFTW With Improved Communications and Framework Optimizations

Chapter 3 presented the implementation of the FFTW library within the new middleware along with a possible optimization. This optimization consists of overlapping the creation of the plan with the transfer of the input data to the server. Results presented in Chapter 5 showed the percentage of reduction in execution time when both proposals (with and without the optimization) are compared. In this appendix, we present the results of the execution time with this optimization of the 1D DFT using double complex data type as input and output.

Figure D.1 shows the estimation of the execution time. Results show an speed up bigger than the attained without the optimization (up to 25x). Finally, Figures D.2(a), D.2(b) and D.2(c) show the breakdown of the total execution time for different system configurations. Notice the reduction in the time for creating the plan thanks to the optimization.

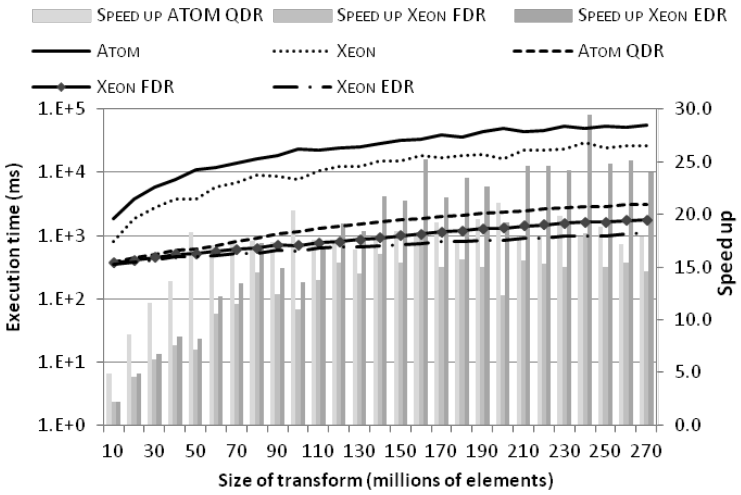
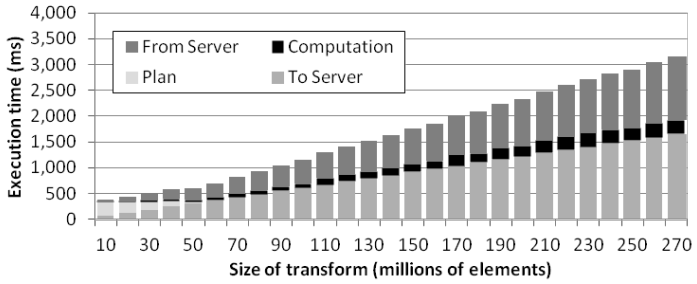
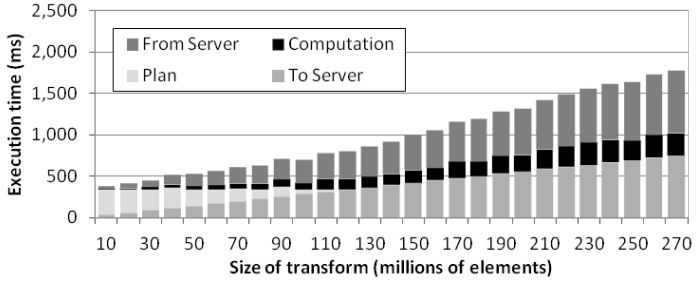


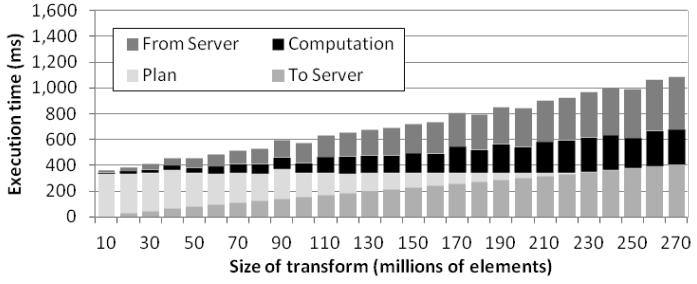
Figure D.1: Estimation of the FFTW 1D in different system configurations using the optimization.



(a) Atom system using QDR network adapter.



(b) Xeon system with FDR network adapter.



(c) Xeon system with EDR network adapter.

Figure D.2: Execution time breakdown for the FFTW 1D function using the optimization.

Bibliography

[1] VMware. VMware Virtualization for Desktop & Server. http://www.vmware.com/, 2015.

[2] Xen. The Xen Project. http://www.xenproject.org/, 2013.

[3] KVM. Kernel-based Virtual Machine. http://www.linux-kvm.org/, 2010.

[4] VirtualBox. Oracle VM VirtualBox. http://www.virtualbox.org/, 2015.

[5] A.A. Semnanian, J. Pham, B. Englert, and Xiaolong Wu. Virtualization Technology and its Impact on Computer Hardware Architecture. In *2011 Eighth International Conference on Information Technology: New Generations (ITNG)*, pages 719–724, April 2011.

[6] Mellanox. ConnectX-3 VPI Single and Dual QSFP+ Port Adapter Card User Manual. http://www.mellanox.com/, 2013.

[7] Intel. Intel Ethernet Server Adapter I350. http://www.intel.com/content/www/us/en/ethernet-controllers/ethernet-i350-server-adapter-brief.html, 2013.

[8] NVIDIA. NVIDIA GRID Technology. http://www.nvidia.com/object/grid-technology.html, 2015.

[9] Jike Song, Zhiyuan Lv, and Kevin Tian. KVMGT: A Full GPU Virtualization Solution. http://www.linux-kvm.org/wiki/images/f/f3/01x08b-KVMGT-a.pdf, 2014.

[10] C. Reaño, F. Silla, A.J. Peña, G. Shainer, S. Schultz, A. Castello, E.S. Quintana-Orti, and J. Duato. Boosting the performance of remote GPU virtualization using

InfiniBand connect-IB and PCIe 3.0. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 266–267, September 2014.

[11] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *Euro-Par 2010 - Parallel Processing*, pages 379–391, August 2010.

[12] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi. DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment. In *2012 SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC)*, pages 1207–1214, November 2012.

[13] S. Iserte, A. Castello, R. Mayo, E.S. Quintana-Orti, F. Silla, J. Duato, C. Reaño, and J. Prades. SLURM Support for Remote GPU Virtualization: Implementation and Performance Study. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 318–325, October 2014.

[14] S. Iserte, J. Prades, C. Reaño and F. Silla. Increasing the Performance of Data Centers by Combining Remote GPU Virtualization with SLURM. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2016)*, May 2016.

[15] NVIDIA. cuBLAS Library 7.0. https://developer.nvidia.com/cuBLAS/, 2015.

[16] Texas Advanced Ccomputing Center. GotoBLAS2. https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2/, 2014.

[17] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 25:1–25:12, 2013.

[18] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180, 2009.

[19] NVIDIA. cuBLAS-XT Library 7.0. https://developer.nvidia.com/cuBLASXT/, 2015.

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

[21] NVIDIA. cuFFT Library 7.0. https://developer.nvidia.com/cuFFT/, 2015.

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

[23] M. Frigo and S.G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, Feb 2005.

[24] Sander Pronk, Szilrd Pall, Roland Schulz, Per Larsson, Pr Bjelkmar, Rossen Apostolov, Michael R. Shirts, Jeremy C. Smith, Peter M. Kasson, David van der Spoel, Berk Hess, and Erik Lindahl. GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29(7):845–854, 2013.

[25] Supermicro. X10 MicroBlade with MBI-6418A-T7H modules. http://www.supermicro.nl/products/MicroBlade, 2015.

[26] iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool. https://github.com/esnet/iperf, 2015.

[27] Antonio J. Pe na, Carlos Rea no, Federico Silla, Rafael Mayo, Enrique S. Quintana-Ortí, and José Duato. A complete and efficient CUDA-sharing solution for HPC clusters. *PARCO Journal*, 2014.

[28] Mellanox. Mellanox OFED GPU Direct RDMA Product Brief. http://www.mellanox.com/, 2014.

[29] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1999.

[30] John W. Tukey James W. Cooley. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965. ISSN 00255718, 10886842. URL http://www.jstor.org/stable/2003354.

[31] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984.

[32] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, 2003.

[33] F. Nammour and N. Mansour. Comparative evaluation of object request broker technologies. In *Computer Systems and Applications, 2003. Book of Abstracts. ACS/IEEE International Conference on*, pages 66–, July 2003.

[34] The DL_POLY Molecular Simulation Package. http://www.ccp5.ac.uk/DL_POLY.

[35] TINKER - Software Tools for Molecular Design. http://dasher.wustl.edu/tinker/.

[36] NVIDIA Jetson TK1 Development Kit. http://developer.nvidia.com/jetson-tk1.