An Efficient Implementation of GPU Virtualization in High Performance Clusters

José Duato[1], Francisco D. Igual[2], Rafael Mayo[2], Antonio J. Peña[1], Enrique S. Quintana-Ortí[2], and Federico Silla[1]

[1] Departamento de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia (UPV), 46022–Valencia, Spain
{jduato,fsilla}@disca.upv.es, apenya@gap.upv.es
[2] Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I (UJI), 12071–Castellón, Spain
{figual,mayo,quintana}@icc.uji.es

Abstract. Current high performance clusters are equipped with high bandwidth/low latency networks, lots of processors and nodes, very fast storage systems, etc. However, due to economical and/or power related constraints, in general it is not feasible to provide an accelerating co-processor –such as a graphics processor (GPU)– per node. To overcome this, in this paper we present a GPU virtualization middleware, which makes remote CUDA-compatible GPUs available to all the cluster nodes. The software is implemented on top of the sockets application programming interface, ensuring portability over commodity networks, but it can also be easily adapted to high performance networks.

Keywords: Graphics processors (GPUs), virtualization, high performance computing, clusters, Grid.

1 Introduction

Virtualization of hardware resources is receiving considerable attention in the last years as a means to reduce the economic cost, ease the administration, and provide better security in large data centers [4].

On the other hand, graphics processors are increasingly being adopted as a hardware solution to accelerate computationally-intensive applications [1,13,14]. Improvements in the programmability of these architectures [12,2] and their excellent performance-power ratio will probably generalize their use in large clusters for high performance computing (HPC) in the near future. However, adding one hardware accelerator to every node in an HPC cluster is not efficient, neither from the performance point of view nor from the power consumption perspective, because, on one hand, not all applications can take advantage of the accelerator and therefore there is no need for a large number of them and, on the other hand, current GPUs have a great impact on the overall power consumption of the system[1]. Economic cost, maintenance, and space also advise

[1] A GPU may well increase the power consumption of an HPC node by 20-30%.

H.X. Lin et al. (Eds): Euro-Par 2009 Workshops, LNCS 6043, pp. 385–394, 2010.

against the one GPU per node solution. Thus, future HPC clusters may well include a few of these accelerators in certain nodes of the system.

In this paper we present a prototype middleware that virtualizes a hardware resource like a GPU in an HPC cluster, as a front-end virtualization. This type of virtualization can be implemented by *device emulation*, that is, by providing a complete replication of the entire hardware accelerator, so that the architecture can be emulated on a different one. However, for computationally-intensive applications this approach is not valid due to the emulation overhead. A better choice to service HPC applications is to offer a virtualized hardware platform, time-sharing the real resource among the users. This is the approach we adopt in our proposal by *Application Programming Interface* (API) *remoting*. Therefore, there is no need of hardware support nor silicon changes.

Our middleware offers the possibility of running different parts of an application on different accelerators, dynamically selecting the most suitable one. Although we have focused only on GPUs, our software can be extended to other types of accelerators. Thus, the goal is to offer virtualized CUDA-compatible GPU devices that can be used by all nodes in the cluster with low overhead. Although similar approaches have been recently followed in the field of virtual machines and graphics [5], the specifics of our target environment and CUDA led us to adopt a different approach since, among others, we do not have to take care of visual output or suspend and resume functionality. Instead, we have to deal with CUDA specifics such as streams and execution control.

The rest of this paper is organized as follows: Section 2 presents the details of the proposed virtualization solution. Section 3 introduces performance related issues. In Section 4 we discuss the current development status of our implementation. Next, Section 5 presents some performance results and, finally, Section 6 summarizes the conclusions of our work.

2 Virtualized GPU Architecture

GPUs are integrated devices in the form of cards that are attached to a server with a general-purpose processor via a PCI-Express (PCIe) bus. To exploit the GPU computing power, part of the program has to be written as a *kernel*, which at runtime is sent and executed on the GPU. The GPU driver is in charge of transferring the program, initiating its execution, and handling its completion.

Both the general-purpose server and the GPU feature separate memory maps. Transferring the data required by the kernel and later retrieving back the results is explicitly addressed by the user. Therefore, in an HPC system where a node without a local GPU is running an application that invokes a GPU kernel, we have to provide support for transferring the kernel and data, and dealing with the initiation/completion of the kernel execution on a remote GPU. In particular, our virtualization middleware consists of two parts: the client middleware is installed as a shared library on all nodes of the HPC cluster which have no local GPU; and the server middleware is executed in the node(s) equipped with GPU(s). We name these nodes hereafter as clients and server(s), respectively.

2.1 Implementation

The current implementation of the virtualization software targets the NVIDIA CUDA programming environment and the NVIDIA G80 and GT200 series.

CUDA enables general purpose computing on the latest NVIDIA GPUs in a C-like programming language, exposing the device architecture as a set of SIMD multiprocessors. This smooths the learning curve for the non-expert programmers on graphics-specific programming languages such as OpenGL and Cg. More detailed information about CUDA can be found in [12].

Client. These nodes employ a library of wrappers to the CUDA Runtime API. During the compilation of the application, two different object files are generated exploiting the compiler driver options: the GPU-module comprises the device code image to be executed on the remote GPU; and the CPU-module contains the code that has to be executed on the local general-purpose processor. The global executable file includes all the functionalities required to identify and connect to the server, locate and send the GPU image file, submit requests for the execution of a specific kernel and transfer dynamic data, and receive and pass back to the application the output resulting from the remote execution.

Server. On the server we add a GPU network service which listens for requests on a TCP port. To deal with the low-level GPU-module operations which are not supported by the CUDA Runtime API, this daemon has been implemented using the low-level Driver API. A library scheme of both client and server applications is shown in Figure 1.

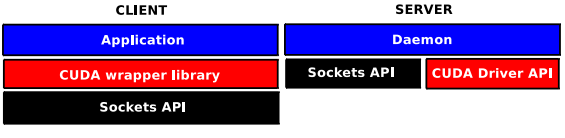

Fig. 1. Scheme for the client and server applications library

The daemon serves requests of CUDA calls on the local GPU. Each remote execution is served by a new process on an independent GPU context. The use of threads for this purpose is not considered an option as potential segmentation faults on Driver API calls could lead to server termination.

In general, the execution of a kernel requires several phases: in the initialization stage, the server receives the GPU code image with the kernels to be executed and the definition of the variables statically allocated by the client application. Once this initialization is completed, the server is able to process a request for executing a kernel. If there is additional data to be used, it must be transferred from client to server before the execution starts. Once the kernel execution is completed, the output data is available to be sent back to the client.

Communication protocol. The data protocol for the communication between client and server has been designed to be as simple as possible, so that communications involve little computation and make an efficient use of network resources. Both data and control flows make use of the network.

Until automatic server discovery is implemented, the first action on the client side is an explicit call to an initialization function, which connects to a specific server. This function automatically locates and sends the application-associated GPU-image file to the server.

In the communication protocol, the first 32 bits of the stream request identify the function which has been called, while the subsequent data is function-dependent, specifying the particular parameters of each function call. The server always sends a 32-bit result code, and possibly more data depending on the requested function.

A sample sequence diagram of the communications generated by a matrix-matrix multiplication execution is shown in Figure 2, which illustrates the following steps:

1. The client application opens a socket connection to the server, where a daemon process is listening. The client then locates and sends the GPU-image to the server, which loads it into a new GPU context. Upon completion, the server sends the result code of the module load operation back.
2. The client requests `memory allocation` on the GPU memory map for the three matrices involved in matrix multiplication. For each one of the three requests, the server replies with the result code of the allocation operation, followed by the pointer to the allocated memory.
3. The next step consists in sending the source data matrices to the GPU memory. To do so, the client sends two `memory copy` requests, each of them specifying the destination pointer, size of the data to be transferred and direction of the copy (from host to device[2]), followed by the corresponding data. Once a request is received and executed, the server sends back the result code of the operation.
4. The GPU is then ready to execute the matrix-matrix multiplication kernel. Next, the client application sends a `launch` request, specifying the kernel to be executed and its execution stack, which consists in a `grid` and `block` configuration, as well as the parameters of the kernel (those commonly required by the BLAS `sgemm` subroutine). Once the launch is done, the server daemon sends back the corresponding result code.
5. At this point, the result of the matrix multiplication is stored in the GPU memory. To transfer it to the local memory, the client application sends a new `memory copy` request, this time specifying the direction as "device to host". The server response is the corresponding result code followed by the requested data (only if the copy operation was successful).
6. Allocated memory is released next. To accomplish this, the client application sends a `free` request per matrix, receiving a result code per request.

[2] In CUDA terminology, `host` stands for a computer holding a CUDA-compatible GPU card, and `device` stands for the GPU itself.

7. The last step consists in calling a `destroy` function, which closes the socket. Upon reception, the daemon server process quits servicing the current execution and releases the associated resources.

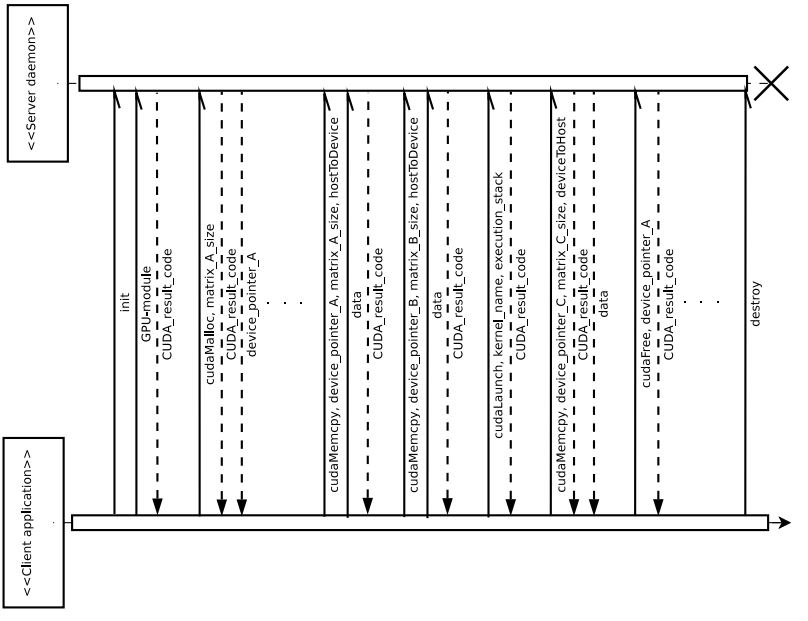

Fig. 2. Matrix-matrix multiplication. Sequence diagram of client-server communications. Memory allocation and release operations are summarized for legibility purposes.

3 Performance Considerations

Virtualizing the GPUs implies an overhead due to the communications over the network, which depends on the specifics of the interconnect (latency and bandwidth). In our approach we intend to explore both the Gigabit Ethernet standard accessed via the sockets API and the new extensions to the HyperTransport (HT) technology, recently proposed in the High Node Count HyperTransport Specification [3]. This technology provides a non-coherent shared memory map for a large number of computing nodes with very low latency, as data transfers are managed by hardware with no intervention from the operating system kernel.

One advantage of the socket-based implementation is that it could be used even in low performance networks for academic purposes, thus offering access to a few high performance GPUs concurrently to all the students. On the other hand, the HT based implementation is expected to offer clients of an HPC cluster seamlessly access to a remote GPU with negligible overhead.

Users of the virtualization middleware must take into account that the usage of asynchronous CUDA calls will notably increase the performance of their applications reducing the communication overhead.

To reduce the response time, our server uses a `prefork` technique as follows:

1. The parent server is started.
2. The parent server creates a child server which will serve all requests from a single remote execution.
3. The child server receives a connection request.
4. The child server communicates this event to its parent.
5. The parent server spawns another child to attend eventual requests.
6. Children terminate after the connection is closed by their respective client.

In addition, another tweak is introduced to save time: before a child server blocks waiting for an upcoming connection request, it pre-initializes the CUDA driver API environment and creates a new context, so it is ready to load a GPU image immediately when it is received.

Finally, to attain high-performance data transmission over a TCP/IP network, Nagles's algorithm [9,10] –TCP layer default congestion control algorithm– has been disabled on both client and server sides. Basically, this algorithm delays the effective sending of TCP frames until a buffer is filled in or a timer expires. This behavior provides good performance in many environments, preventing the transmission of a large number of small packets, which would waste most of the network bandwidth transmitting packet headers. However, in HPC a precise control of the moment a frame must be sent out is desired. In Linux operating system (OS), this is achieved by explicitly choosing the time when the TCP transmission buffer must be flushed by managing TCP layer socket options and policies. A more detailed discussion about Nagle's algorithm can be found in [7].

4 Development Status

This project is, at the moment of the writing, ongoing so this could be considered a proof of concept.

We concentrate our development efforts on the TCP based approach, but we expect to adapt the developed middleware to the HT based interconnect by just changing the communication routines to send and receive data over the network.

4.1 Implemented Functionality

Thus far we have successfully implemented on both client and server sides the following of the CUDA Runtime API:

- Device Management Runtime.
- Thread Management Runtime.
- Event Management Runtime.
- Execution Control Runtime.
- Part of the Memory Management Runtime.
- Error Handling Runtime.

This subset of the API is sufficient to build a series of commonly used applications and obtain some timing results, which are presented in Section 5.

4.2 Future Work

We are working on the completion of the whole CUDA Runtime API. In particular, current missing functionalities comprise:

- Stream Management Runtime.
- Texture Reference Management Runtime.
- Part of the Memory Management Runtime, mainly asynchronous operations.

One drawback of the current implementation is that it needs to keep the CPU and GPU codes in separated files. The GPU code is compiled with the NVIDIA compiler driver `nvcc` using its "device code repositories" feature (see [11]) to obtain the GPU-image file. On the other hand, the CPU code is compiled using a C or C++ compliant compiler (such as GNU or Intel C Compilers) to obtain the final executable. This leads to the unavailability of the `CUDA C language extensions` (such as the simplified kernel call syntax) on host code. This separation is mandatory because during compilation of mixed GPU and CPU code, `nvcc` automatically inserts calls to undocumented CUDA Runtime API library functions, (presumably to allow the application locate the embedded GPU code in the executable, among others). To address this, we will develop a preprocessor which will transparently take care of code separation and compilation.

Another limitation of the mandatory code separation step is the impossibility of using prebuilt CUDA libraries such as CUBLAS, because at the moment our runtime is unable to locate embedded GPU code. However, we expect to overcome this problem because GPU code is easily recognized in the executable.

On the final stage of the development, we will deal with multi-server related functionalities, such as automatic discovery and load balancing.

Additionally, we will explore more network related tweaks, such as TCP `defer accept` and `quick ack` options, and we will adapt our communication routines to the high performance HT based network.

When completed, we may consider adapting the implementation to Windows OS based systems, and the recently emerged OpenCL framework [8].

Finally, in the long term we intend to generalize our implementation to different kinds of accelerators.

5 Results

In this section we evaluate the impact of the virtualization overhead, using two case studies: the product of two matrices and the Fast Fourier Transform (FFT).

The nodes of the cluster employed in the evaluation are equipped with two Quad Core Intel® Xeon® E5410 processors (2.33 GHz, 8 GB RAM), running the Linux OS (kernel 2.6.18). The node interconnect is a Gigabit Ethernet. The GPU is an NVIDIA Tesla C1060 (driver version 180.22) attached to a PCIe 2.0 x16 port[3]. The server daemon has been built over CUDA Toolkit 2.1.

[3] A PCIe 2.0 x16 graphics link features a maximum bandwidth of 8 Gbytes/s.

The matrix-matrix product implemented in routine `sgemm` as part of Intel MKL (v10.1) is employed on the CPU. On the GPU, we have used Volkov's implementation of the matrix-matrix product routine [15], as this is currently the base for the tuned implementation in CUBLAS 2.1. On the other hand, we have used the FFTW library (v3.2) on the CPU and Volkov's FFT implementation on the GPU, over 1024 complex single precision points. To exploit the GPU massive parallel capabilities, our tests comprise different number of FFT operations, provided that GPU is able to compute multiple FFTs in parallel. To accommodate network variability times are averaged over 30 executions.

The left-hand plot in Figure 3 shows that the execution of Volkov's kernel on a virtualized GPU over small and moderate-size matrices is slightly slower than the local CPU implementation in MKL –maximum of 2.5 secs. on a $8,192 \times 8,192$ matrix–, while for large matrices it is up to a 15% faster, saving 10.6 secs. The figure also shows that most of the time is spent in memory transfers, due to network bandwidth limitations. The right-hand plot in the same figure reports that the execution times for the FFT are between 150 and 1,150 msecs. faster in the local CPU than in the remote GPU, once more due to the network limitations.

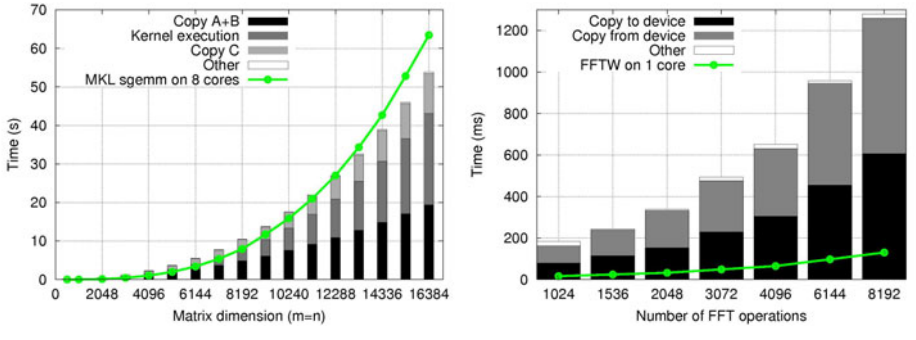

Fig. 3. Left: `sgemm` processing times on CPU vs. virtualized GPU; "Other" includes initialization, memory allocation, memory release and destruction operations. **Right:** `FFT` processing times; "Other" also includes kernel execution.

This result demonstrates that computing a matrix multiplication over a remote GPU can be faster than doing it over the local CPU, even when the connection happens to be a commodity network. However, when the problem requires less computation per data, as is the case for the FFT (the cost of the matrix multiplication is $O(n^3)$ while that of the FFT is $O(n \log n)$, where n is the problem size), there is a serious bottleneck in low bandwidth networks.

A comparison of the execution times of the CUDA function call of our implementation with those of a "local" CUDA execution, reveals that all the former functions are slightly slower (around 10 ms. per call), except for memory copies when a large amount of data is sent over the network (see the plots in Figure 4). In particular, remote copies are around 15 times slower than local ones, yielding a maximum overhead of 28 seconds for the largest problem on `sgemm`.

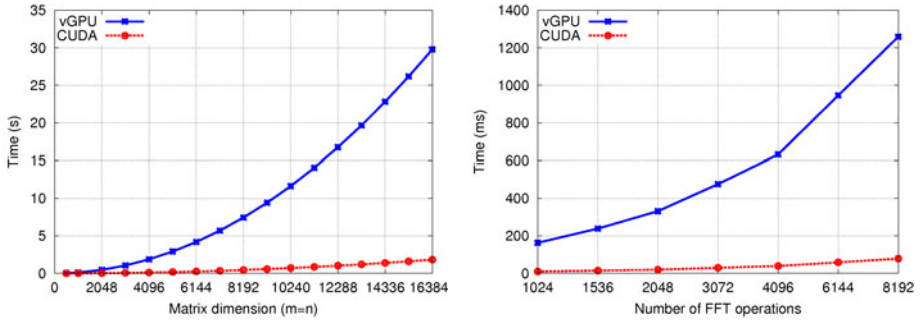

Fig. 4. Left: Execution time of the three matrix copies involved in the matrix-matrix multiplication. **Right:** Time for the memory copy operations (both directions) of FFT.

Those results illustrate that the overhead introduced by our implementation is mostly caused by network related delays, as PCIe bandwidth is an order of magnitude faster than the Gigabit Ethernet one. Therefore, we expect to reach a performance that is close to that obtained with a local GPU execution when the target network is based on HT, as this network will attain 3.2 Gbytes/s, which is –according to our tests– around a half of the effective peak bandwidth of the GPU reads through the PCIe bus. Furthermore, the latency estimations for the HT-based network are around a few μsecs. [6], which is an order of magnitude lower than the latency for a TCP frame on the network used in our tests.

6 Conclusions

We have implemented a GPU virtualization prototype which enables seamlessly remote CUDA Runtime API calls. The middleware enables an efficient use of an HPC cluster where only some of the nodes are equipped with accelerators.

We have shown that our approach can deliver reasonable performance for clusters connected via a commodity network. As a major part of the time is spent on communications, we expect a negligible degradation in performance in case the nodes are connected via a high performance network.

There is much future work in completing the whole CUDA API and solving multi-server related issues. Eventually, we also expect to generalize this solution to OpenCL compatible accelerators.

Acknowledgements

Researchers at UPV were supported by PROMETEO from Generalitat Valenciana under Grant PROMETEO/2008/060.

Researchers at UJI were supported by the Spanish Ministry of Science and FEDER (contract no. TIN2008-06570-C04-01), and by the Fundación Caixa-Castelló/Bancaixa (contract no. P1B-2007-19).

References

1. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S.: Solving dense linear systems on graphics processors. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 739–748. Springer, Heidelberg (2008)
2. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: stream computing on graphics hardware. In: SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, pp. 777–786. ACM, New York (2004)
3. Duato, J., Silla, F., Yalamanchili, S., Holden, B., Miranda, P., Underhill, J., Cavalli, M., Brüning, U.: Extending HyperTransport protocol for improved scalability. In: Proceedings of the First International Workshop on HyperTransport Research and Applications (WHTRA 2009), pp. 46–53 (2009)
4. Figueiredo, R., Dinda, P.A., Fortes, J.: Guest editors' introduction: Resource virtualization renaissance. Computer 38(5), 28–31 (2005)
5. Andres Lagar-Cavilla, H., Tolia, N., Satyanarayanan, M., de Lara, E.: VMM-independent graphics acceleration. In: VEE '07: Proceedings of the 3rd international conference on Virtual execution environments, pp. 33–43. ACM, New York (2007)
6. Litz, H., Froening, H., Nuessle, M., Bruening, U.: VELO: A novel communication engine for ultra-low latency message transfers. In: ICPP '08. 37th International Conference on Parallel Processing, September 2008, pp. 238–245 (2008)
7. Mogul, J.C., Minshall, G.: Rethinking the TCP nagle algorithm. Computer Communication Review 31(1), 6–20 (2001)
8. Munshi, A. (ed.): OpenCL 1.0 Specification. Khronos OpenCL Working Group (2009)
9. Nagle, J.: Congestion control in IP/TCP internetworks. Computer Communication Review 14(4), 11–17 (1984)
10. Nagle, J.: RFC 896: Congestion control in IP/TCP internetworks (January 1984)
11. NVIDIA: Nvidia CUDA Compiler Driver NVCC. NVIDIA (2008)
12. NVIDIA: Nvidia CUDA Programming Guide Version 2.1. NVIDIA (2008)
13. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26(1), 80–113 (2007)
14. Stone, S.S., Haldar, J.P., Tsao, S.C., Hwu, W.-m.W., Liang, Z.-P., Sutton, B.P.: Accelerating advanced MRI reconstructions on GPUs. In: CF '08: Proceedings of the 2008 conference on Computing frontiers, pp. 261–272. ACM, New York (2008)
15. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Piscataway, NJ, USA, pp. 1–11. IEEE Press, Los Alamitos (2008)