Project 9e
vCUDA Framework Development
for
GPU Virtualization

Name	Roll No	Contact No	Email
Aashish Chauhan	MT2011003	9620809779	aashish.chauhan@iiitb.org
Aditya Swaminathan	MT2011006	9164948286	adityats@iiitb.org
Bakul Mittal	MT2011025	9686050181	Bakul.Mittal@iiitb.org
Padmaraj R	MT2011098	8095094622	Padmaraj.R@iiitb.org
Sridutt Nayak	MT2011154	9901216207	sriduttv.nayak@iiitb.org

Team Leader : Aditya Swaminathan

Contents

1 Introduction

System level virtualization technology is spawned from the continued growth in hardware performance and increased demand for high performance computing in various arenas. VMM(Virtual Machine Manager) technology has been applied to many areas including intrusion detection, high performance computing and device driver reuse. Here we have to develop a vCUDA (Virtual CUDA) for High Performance Computing on graphics processing units (GPUs). This uses hardware acceleration provided by GPUs to address the performance issues associated with VMs.

2 Problem description

The virtualization of the hardware (e.g. CPU, servers and peripheral devices) is prevalent but the limiting factor in the GPU virtualization is the lack of standard interface at the hardware level. This can be overcome in two different ways,

1. Request from guestOS (guest Operating System) is forwarded to hostOS using software emulation.

2. Intercept the graphics protocol stream and redirect it to the hostOS which requires us to replace the dynamic link libraries with the protocol stubs.

3 CUDA

3.1 What is CUDA?

Computing is evolving from "central processing" on the CPU to "co-processing" on the CPU and GPU. CUDA or Compute Unified Device Architecture is a parallel computing architecture developed by nVIDIA.[5]

CUDA is the computing engine in nVIDIA GPUs that is accessible to software developers through variants of industry standard programming languages.

CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. Unlike CPUs however, GPUs have a parallel throughput architecture that executes several concurrent threads slowly, rather than executing a single thread very quickly. This approach is used for solving non graphical tasks on GPUs and

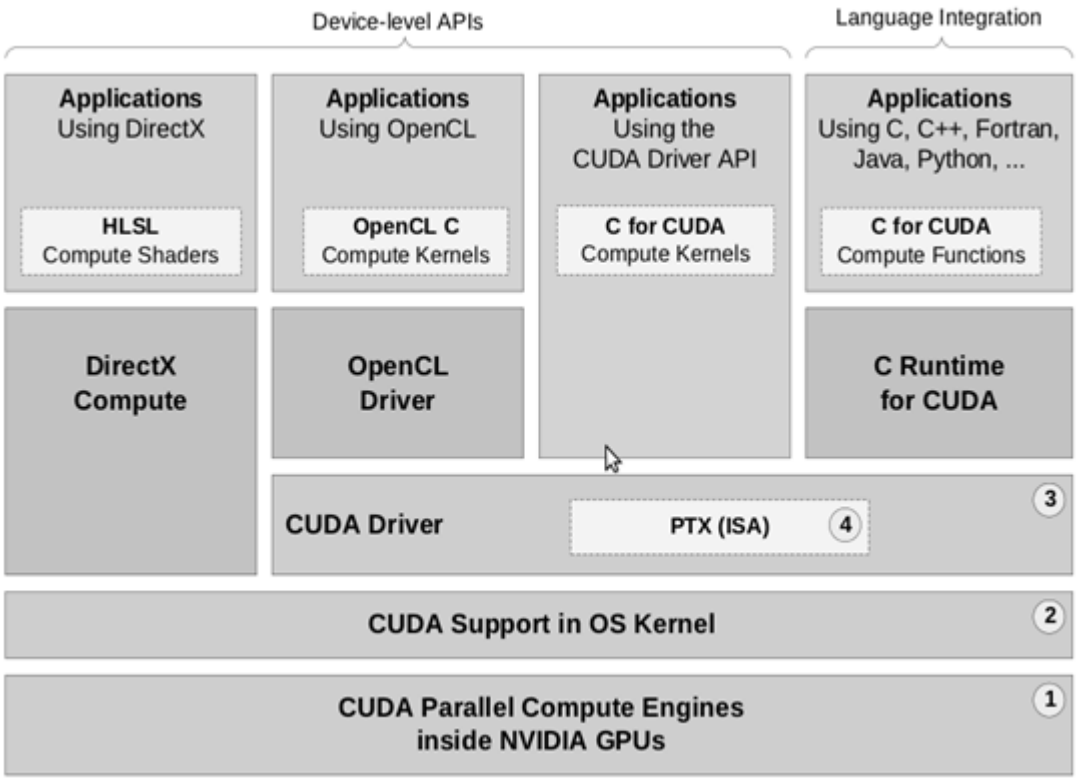

Figure 1: CUDA Architecture[6]

this paradigm is known as General-purpose computing on graphics processing units (GPGPU).

3.2 CUDA Architecture

The CUDA architecture consists of the following components:[3][6]

1. Parallel compute engines inside nVIDIA GPU's.
2. OS kernel-level support for hardware initialization, configuration, etc.
3. User-mode driver, which provides a device-level API(Application Programming Interface) for developers.
4. PTX instruction set architecture (ISA) for parallel computing kernels and functions.

3.3 Working of CUDA Compiler

CUDA toolkits enables parallel processing in nVIDIA GPU through remote procedure calling. Source files for CUDA applications consist of a mixture of conventional C++ 'host' code, plus GPU 'device' functions. The CUDA compilation trajectory separates the device functions from the host code.The nVIDIA code complier compiles the device functions and embeds the compiled GPU functions as load images in the host object file. Also, the CUDA compilation process involves compilation of the host code using any general purpose C/C++ compiler that is available on the host platform.In the linking stage, specific CUDA runtime libraries are added for supporting remote SIMD procedure calling and for providing explicit GPU manipulation such as allocation of GPU memory buffers and host-GPU data transfer.[4]

3.4 CUDA Software Development Environment

The CUDA Software Development Environment supports two different programming interfaces:

1. A device-level programming interface, in which the application uses DirectX Compute, OpenCL or the CUDA Driver API directly to configure the GPU, launch compute kernels, and read back results.

   When using the device-level programming interface, developers write compute kernels in separate files using the kernel language supported by their API of choice.

2. A language integration programming interface, in which an application uses the C Runtime for CUDA and developers use a small set of extensions to indicate which compute functions should be performed on the GPU instead of the CPU.

   When using the language integration programming interface, developers write compute functions in C and the C Runtime for CUDA automatically handles setting up the GPU and executing the compute functions. This programming interface enables developers to take advantage of native support for high-level languages such as C, C++, Fortran, Java, Python, reducing code complexity and development costs through type integration and code integration.

3.5 CUDA Software Stack

The CUDA software stack is composed of several layers. Figure 2 illustrates the components of CUDA and how an application interfaces with a CUDA

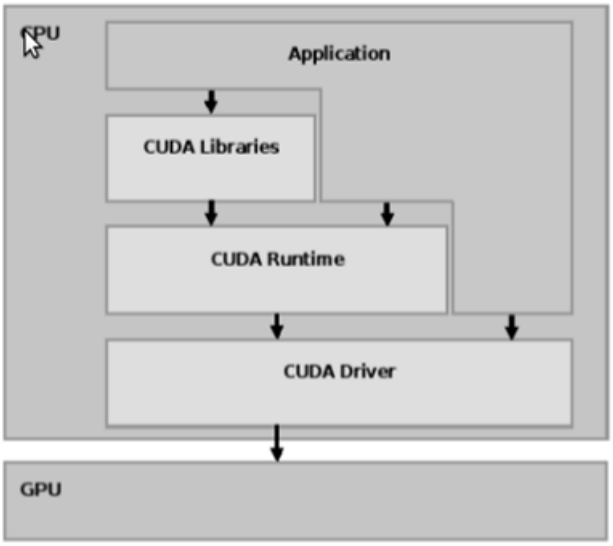

Figure 2: CUDA Software Stack

enabled device.

At the lowest level a CUDA enabled GPU must be present in the system. In between the device and CUDA application is the CUDA device driver. The CUDA device driver is distinct from a graphics driver. It provides an interface to the GPU′s general purpose capabilities and is not required if the device is to be used solely for the rendering of graphics.

4 vCUDA

4.1 vCUDA Library

vCUDA library resides in the guestOS , it intercepts the API calls and redirects it to the stub. Runtime API will be used for virtualization which is a recommended interface for the programmers. There are totally 56 runtime + 6 internal APIs and these do not interact with other APIs. So, we just wrap them up and forward it to the stub as their logic may change as the GPU provider decides upon. NVCC (compiler for CUDA) generates intermediate code combined with control flow analysis to customize the virtual logic for each API. These intercepted API calls are packed into a global queue. These contain copy of the arguments to the corresponding functions as well as an opcode.[7]

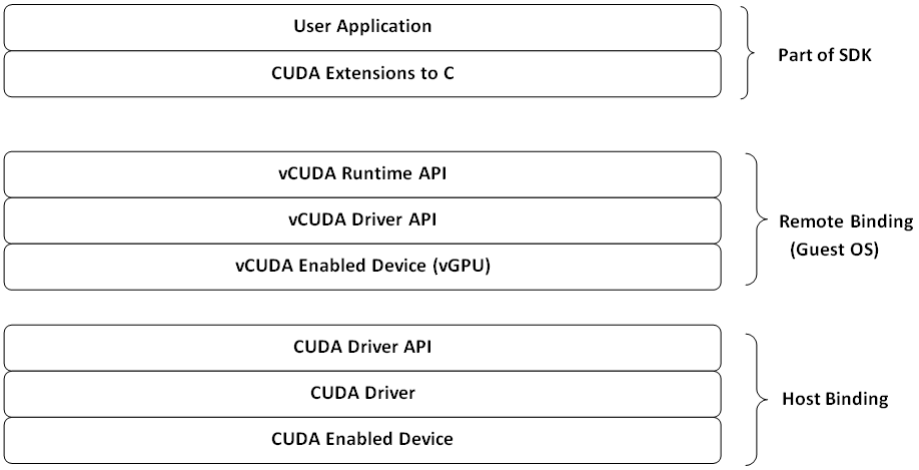

Figure 3: vCUDA architecture

4.2 vGPU

vGPU is used by vCUDA libraries and is represented as a large data structure in memory maintained by vCUDA library. It has three main functions, first to abstract the features of the GPU and give each application a complete view of the underlying hardware; it creates a GPU context for every vCUDA application running on top of it. Second, local device memory management, when CUDA application allocates memory as per the request of the CUDA application, it sends a request to the stub to reserve actual memory for the application. Also, the vGPU has to maintain the mappings of the local and remote address to prevent the unnecessary memory copies and leaks. vGPU stores the CUDA API flow, most of the API's opcode and parameters are stored in a global queue in memory or a file of a file system.

4.3 vCUDA Stub

vCUDA receives and interprets remote requests, it creates the execution context for the API calls and returns results to the guestOS. It manages the actual hardware resource allocations, matches API call parameters and keeps a consistent view of the states. One thread is maintained for each client in orderâĂŃ to receive the CUDA commands via RPC(Remote Procedure Call) channel and execute those commands on behalf of the client applications.

4.4 Lazy RPC

To facilitate proper computation there is a necessity of a high level communication between hostOS and guestOS. So, we fall back to XML-RPC. Lazy RPC is a modified form of this RPC which helps us to have an efficient communication. Each API call redirect may take up very high number of context switching which is a costly computation by itself. So, we create the batch file kind of an approach where many of the calls can be redirected.

4.5 Multiplexing

Any device must be enabled to operate with multiple hosts, which means we have to have a mechanism where the GPU multiplexing in application level through co-operation of the vCUDA libraries. One thread mapped to single device and multiple threads mapped to single device conditions would be explored.

5 Architecture for GPU Virtualization

Our proposed system would have a CUDA application running in its VM, currently this would not be allowed as the VM cannot access the hardware (GPU). Front-end of our system will comprise of a vCUDA library and a vGPU which helps the CUDA application running on the system to see a GPU accessible for it. This framework will create the memory space when requested by the CUDA Application. Xen Loop communication working on the base of a linux kernel will be used for RPCs[2]. The back-end of our system or the stub will unwrap the request from the channel and allocate the memory and carry out the operations defined in the opcode on the data.[1]

1. Front-end
   (a) vCUDA Library, collecting API Calls and receive the data as well as the commands to be executed on the data.
   (b) vGPU, this will create the virtual environment for the CUDA Application to give a feel of the GPU. Packs the data and sends it down the channel.
2. Communication channel (Provided on the Hypervisor) will carry the RPCs that are exchanged between the front-end and the back-end. We will use RPC or Lazy RPC depending on how often we have to communicate the command and the control data.

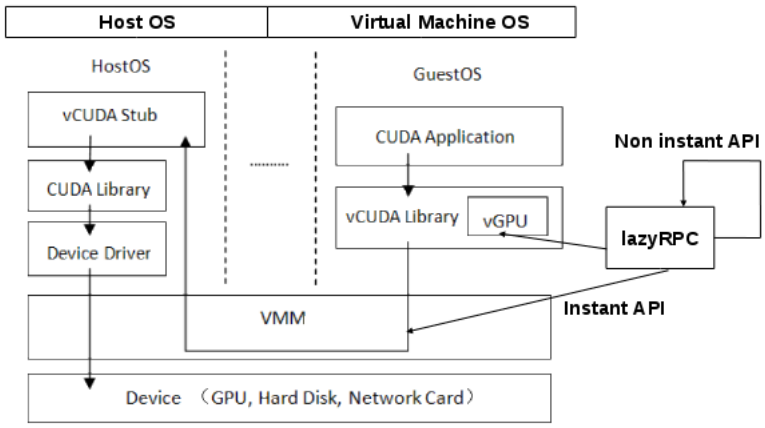

Figure 4: vCUDA data & control flow[7]

3. Back-end (Stub)
   (a) API Call data, parameters and commands are unpacked.
   (b) GPU is being accessed, memory is allocated in the actual GPU and the API calls are redirected. The actual function implementation is carried out on the GPU and the result is received back.
   (c) The result is wrapped back and sent back over the channel.

6 Development Plan

The development plan below indicates the list of activities along with the associated deliverables. Also listed are the corresponding technologies used to achieve the task

1. Install Xen Hypervisor on a CUDA equipped system, understanding CUDA and knowing the vendor specific CUDA drivers. Learn how Hypervisor works and interacts with the different operating systems running in the system.

2. Understand the vCUDA libraries, comprehend APIs and RPCs. Learn how the CUDA application works in a native OS, how it makes the API calls and how the API calls are handled by a GPU.

3. Capture the API calls and its parameters and write them into a file, build a simple vCUDA library. Understand and implement vGPU.

4. Capture the API calls and redirect the parameters and the data through RPC to a native OS and write from these calls into a file. Also implement memory management through vGPU

5. Develop a stub on hostOS which can capture the imformation from the RPC calls from vCUDA library to vCUDA stub and perform the operation on the actual GPU.

7 System Requirements

7.1 Hardware Requirements

- System with CUDA equipped GPU
- 6GB plus the required disk space recommended by the guest operating system per guest. For most operating systems more than 6GB of disk space is recommended.
- One processor core or hyper-thread for each virtualized CPU and one for the hypervisor
- 2GB of RAM plus additional RAM for virtualized guests.

7.2 Software Requirements

- Ubuntu 10.04 or higher for installing Xen Hypervisor
- Any GNU/Linux operating system for development
- Xen Hypervisor 4.1.2
- GPU Drivers and CUDA Libraries

8 Timeline

Date	Task
January 17	Proposing the initial draft of the project and choosing of team leader.
January 27	Final draft of goal statements, SVN repositories set up.
February 7	Brief presentation of the project architecture and plans. Installation and analysis of Xen Hypervisor on a CUDA equipped system.
February 19	Interception of the API calls and store API calls, parameters in a file.
March 1	Mid term review of the project with a working prototype of vCUDA library and conceptual model of a vGPU.
March 16	Modeling and implementing a vGPU
March 30	Developing a vCUDA stub on host OS
April 3	Beta release and submission of first draft of project report
April 19	Final testing and review of project completion. Submission of necessary documents and reports.

References

[1] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. GViM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, pages 17–24, New York, NY, USA, 2009. ACM.

[2] Citrix Systems Inc. What is Xen Hypervisor? `http://www.xen.org/files/Marketing/WhatisXen.pdf`, 2007.

[3] nVIDIA Corporation. *nVIDIA CUDA Architecture*. nVIDIA Corporation, April 2009.

[4] nVIDIA Corporation. The CUDA Compiler Driver NVCC. `http://www.clear.rice.edu/comp422/resources/cuda/nvcc_2.3.pdf`, July 2009.

[5] nVIDIA Corporation. What is CUDA? `http://developer.nVIDIA.com/what-cuda`, January 2012.

[6] Massimiliano Piscozzi. CUDA Architecture. Universita degli Studi di Milano Milan, June 2008.

[7] Lin Shi, Hao Chen, and Jianhua Sun. vCUDA: GPU accelerated high performance computing in virtual machines. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, volume 0, pages 1–11, Los Alamitos, CA, USA, May 2009. IEEE.