UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
ENGENHARIA DA COMPUTAÇÃO

ALAN DIEGO SANTOS

VirtCUDA: Possibilitando a execução de aplicações CUDA em Máquinas Virtuais

Trabalho de Conclusão apresentado como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Prof. Dr. Alexandre da Silva Carissimi
Orientador

Porto Alegre, dezembro de 2010

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Santos, Alan Diego

VirtCUDA: Possibilitando a execução de aplicações CUDA em Máquinas Virtuais / Alan Diego Santos. – Porto Alegre: PPGC da UFRGS, 2010.

35 f.: il.

Trabalho de Conclusão (bacharel) – Universidade Federal do Rio Grande do Sul. Engenharia da Computação, Porto Alegre, BR–RS, 2010. Orientador: Alexandre da Silva Carissimi.

1. Virtualização. 2. GPU. 3. GPGPU. 4. Virtualbox. 5. HGCM. 6. Máquinas virtuais. I. Carissimi, Alexandre da Silva. II. Título.

“Não tentes ser bem sucedido.
Tente antes ser um homem de valor”
— ALBERT EINSTEIN

AGRADECIMENTOS

Gostaria de agradecer primeiramente a minha mãe e meu pai por me apoiarem nos momentos díficeis e complicados, bem como o apoio recebido durante o curso de formação e escrita deste trabalho. Agradeço também ao meu irmão, pelo apoio nos momentos complicados, e minha namorada, por me apoiar durante a escrita deste trabalho. Além disso, gostaria de agradecer a meus amigos por me levarem para as festas e coisas afins, de modo que eu pudesse relaxar da correria que é um curso de engenharia.

Gostaria de agradecer ao Grupo de Redes da UFRGS, em principal ao Luciano Gaspary, Lisandro Zambenedetti, Guilherme Machado, Weverton Cordeiro, Juliano Wickboldt e Roben Lunardi, pelo impulso inicial em minha carreira acadêmica, bem como a "paciência"para explicar e discutir os tópicos muitas vezes. E agradeço principalmente ao professor Alexandre Carissimi que me ajudou muito na confecção deste trabalho, cobrando minha atenção e dedicação em momentos chaves.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

ABI	Application Basic Interface
API	Application Programmer Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Archicteture
GPU	Graphics Processing Unit
HGCM	Host-Guest Communication Manager
HGSMI	Host-Guest Shared Memory Interface
LDU	Load/Store Unit
OS	Operating System
SFU	Special Functions Unit
SIMD	Single Instruction, Multiple Data
SM	Streaming Multiprocessor
VM	Virtual Machine

RESUMO

Atualmente, os computadores são bastante utilizados para aplicações críticas nas organizações. Algumas dessas aplicações utilizam uma grande quantidade de recursos. Contudo, é possível observar recursos subutilizados dentro da infraestrutura dessas organizações. A utilização de máquinas virtuais pode contornar esse problema, uma vez que elas garantem uma maior utilização dos recursos disponíveis em cada máquina física. Todavia, a aplicação dessa técnica em alguns dispositivos é bastante limitada.

Além disso, com o crescimento do poder computacional desses dispositivos, a GPU começou a ser utilizada para a execução de aplicações genéricas. No entanto, para utilizar esses dispositivos era necessário adequar os dados para matrizes, o que dificultava o desenvolvimento de aplicações. Nesse contexto, existem *frameworks* que abstraem essa limitação das GPUs, possibilitando a aplicação do dispositivo em aplicações genéricas, técnica conhecida como GPGPU.

A aplicação de GPGPU em máquinas virtuais é bastante limitada atualmente. Por esse motivo, o objetivo deste trabalho é apresentar uma arquitetura de virtualização da GPU, através da disponibilização do *framework* CUDA para aplicações executando dentro da VM. Além disso, é apresentada uma forma de implementação dessa funcionalidade em uma ferramenta de virtualização, o *VirtualBox OSE*.

Palavras-chave: Virtualização, GPU, GPGPU, virtualbox, HGCM, máquinas virtuais.

ABSTRACT

Today computers are widely used for critical applications in organizations. Some of these applications use a lot of resources. However, it is possible to observe underutilized resources within the infrastructure of these organizations. The use of virtual machines can work around this problem, since they ensure a greater use of available resources on each physical machine. However, applying this technique in some devices is fairly limited.

Furthermore, with the growth of computational power of these devices, the GPU are used for the implementation of generic applications. However, the use of these devices need the adjust the data for arrays, which complicates the application development. In this context, there are some frameworks that abstract the limitation of the GPU, allowing the use of the device in generic applications, a technique known as GPGPU.

The application of GPGPU in virtual machines is currently quite limited. Therefore, the aim of this paper is to present a virtualization architecture of the GPU, by providing the framework for CUDA applications running inside the VM. Furthermore, we present a way to implement this functionality in a virtualization tool called VirtualBox OSE.

Keywords: virtualization, GPU, GPGPU, virtualbox, HGCM, virtual machines.

1 INTRODUÇÃO

Atualmente, com a busca de qualidade e rapidez no desenvolvimento de produtos, a indústria investe pesadamente no melhoria de simuladores. Por esse motivo, os modelos matemáticos que alimentam essas ferramentas têm que ser constantemente aprimorados, o que acarreta em um aumento na sua complexidade e, consequentemente, da demanda de poder computacional. Dessa forma, o ambiente de execução dessas aplicações deve ser extremamente ágil e dispor de vários recursos, possibilitando que essas simulações obtenham o máximo de precisão possível e sejam executadas em tempo hábil.

Usualmente, uma das plataformas empregadas para a execução de aplicações que exijam um alto poder computacional são os *clusters* (RODRIGUES; GOUVÊA; PEREIRA, 2009). Os *clusters* são formados por máquinas *de prateleira (off the shelf)*, o que diminui o custo de sua implementação, além de aumentar sua confiabilidade e disponibilidade. Por outro lado, essa solução apresenta uma série de desvantagens. As máquinas de um *cluster* são dedicadas para esse fim. Dessa forma, caso não existam tarefas em um determinado momento, os recursos não podem ser alocados pelos usuários para a execução de aplicações pessoais (GOLDCHLEGER et al., 2004). Um *cluster* também demanda um ambiente extremamente controlado e estável.

Complementando a situação, pode-se observar uma enorme quantidade de recursos mal utilizados nas instituições. Esses recursos estão disponíveis, por exemplo, nas máquinas pessoais dos funcionários. Ao analisarmos o tempo de utilização dos equipamentos citados, observa-se que eles permanecem inutilizados na maior parte do tempo, principalmente durante a noite. Mesmo durante o expediente, os recursos dessas máquinas *não são totalmente utilizados*. Esse cenário se contradiz com a atual demanda por recursos em outros segmentos das organizações.

Analisando esse contexto, é possível averiguar que a necessidade por recursos e seu desperdício podem coexistir em uma mesma instituição. Isso implica perdas à organização, dado a necessidade de aquisição de recursos para as áreas necessitadas e a existência de recursos que poderiam ser alocados para essa finalidade já presentes na própria infraestrutura da instituição. Vários métodos para o gerenciamento de recursos foram propostos e estudados. Esse tipo de gerenciamento esbarra em problemas de alocação de recursos e escalonamento de tarefas. Por exemplo, os recursos disponibilizados em cada nodo pelos usuários e o salvamento do contexto de execução são alguns dos principais problemas enfrentados.

Uma das formas para contornar os problemas mencionados é a utilização de máquinas virtuais (VMs - *Virtual Machines*). As VMs são máquinas emuladas através de *software*, onde essa executa tarefas como se fosse uma máquina real. Desse modo, as VMs gerenciam a alocação dos recursos por cada máquina virtual. As VMs também possibilitam o total desacoplamento entre o sistema operacional nativo, que executa sobre a máquina

real, e o sistema operacional convidado, que roda sobre a máquina virtual. Desse modo, é possível a execução de diversas máquinas virtuais sobre o mesmo equipamento físico, possibilitando um melhor compartilhamento de recursos entre as diversas tarefas executadas simultâneamente.

Entretanto, a aplicação de determinadas técnicas na utilização de máquinas virtuais ainda é limitado. Um exemplo disso, é a utilização de *GPGPU* (*General-Purpose computing on Graphics Processing Unit*) ou seja, a utilização de Unidades de processamento gráfico (GPUs - *Graphics Processing Units*) na execução de operações genéricas. As GPUs são unidades dedicadas a execução de cálculos gráficos, como a renderização de *pixels*, servindo como aceleradores nesse processo. O paralelismo requerido por esse tipo de aplicação sobrecarrega os processadores genéricos. Dessa forma, as GPUs surgiram para obter desempenho através da exploração dessa característica, executando a mesma operação sobre um grande conjunto de dados. As GPUs evoluíram de rígidos *pipelines* gráficos para arquiteturas programáveis e flexíveis, devido a exigência por gráficos mais realistas. O desempenho com a utilização desse tipo de arquitetura, em determinadas aplicações, é superior ao visto com a utilização de processadores genéricos. Na Figura 1.1, pode-se verificar a diferença entre o tempo de execução entre um processador de uso genérico, um *Intel D*, e uma GPU, um *Tesla 8-series* da Nvidia, na análise química de substâncias. Percebe-se uma *diferença na ordem de grandeza* entre os tempos medidos.

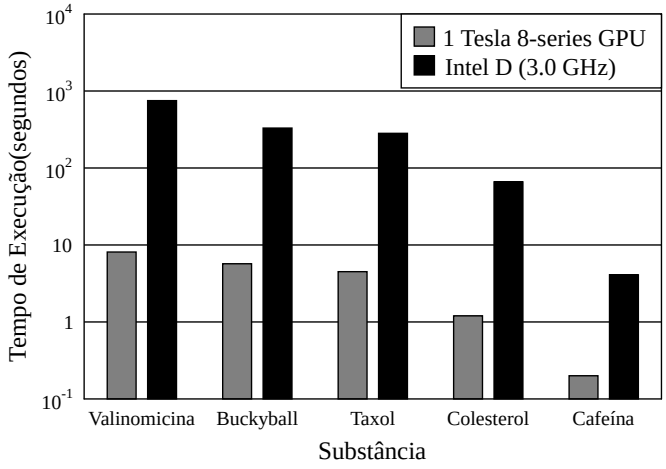


Figura 1.1: Comparação entre o tempo de execução de uma análise química em um processador genérico e uma GPU

1.1 Objetivos

O principal objetivo deste trabalho é possibilitar o acesso e a utilização de GPUs apartir de aplicações que executam na máquina virtual. O segundo objetivo é avaliar o impacto introduzido pela virtualização no desempenho da utilização desse dispositivo.

Para atingir esses objetivos, serão estudados aspectos relacionados a arquiteturas de GPU compátiveis com CUDA, máquinas virtuais e o uso de aplicações e *benchmarks* na avaliação de desempenho. A escolha pelo CUDA foi devido a sua maturidade e forte dissiminação no mercado.

1.2 Organização do trabalho

Este trabalho está dividido em 6 capítulos, separados por área relacionada. No próximo capítulo, a arquitetura e funcionamento da GPU são estudados. No capítulo 3, os conceitos referentes a virtualização são descritos, bem como sua aplicação nos ambientes organizacionais. Por sua vez, o capítulo 4 descreve sobre a forma de implementação da solução proposta e sua implantação em uma ferramenta de virtualização. No capítulo 5, a implementação da solução é avaliada, de modo a determinar o impacto no desempenho da execução de aplicações. Por fim, o capítulo 6, onde são apresentados alguns trabalhos futuros.

2 UNIDADE DE PROCESSAMENTO GRÁFICO

Para a execução das primeiras aplicações gráficas, utilizava-se o mesmo ambiente de *softwares* comuns, ou seja, essas aplicações eram executadas sobre processadores genéricos, também denominados CPUs (*Central Processing Unit*). Contudo, esse tipo de aplicação logo evoluiu a ponto de sobrecarregar os processadores genéricos. Por esse motivo, surgiram os primeiros *chips* acelerados gráficos, denominados GPUs (*Graphics Processing Unit*). Esses dispositivos aproveitavam o paralelismo das aplicações gráficas através da utilização de vários núcleos de processamentos, denominados *processing cores*, ou simplesmente, *cores*.

Primeiramente, a arquitetura desses *chips* era baseada em rígidos *pipelines* gráficos, de modo a aproveitar o paralelismo das operações sobre os *pixels*. Com o desenvolvimento das aplicações e a melhoria no processo de fabricação, várias outras operações foram incorporadas ao conjunto de instruções das GPUs, de modo a agilizar a execução de aplicações gráficas, podendo-se citar as operações sobre ponto flutuante como exemplo de migração. Com o aumento na demanda por gráficos mais realistas, foi necessário a implementação de algoritmos de rasterização (PURCELL et al., 2005), criando um conjunto de operações denominado *shaders* (PHARR; FERNANDO, 2005).

O crescimento da capacidade das GPUs e a melhoria nas técnicas de paralelização de aplicações despertou a possibilidade da utilização desses processadores para a execução de aplicações genéricas. Aplicações extremamente paralelizáveis, como o processamento de imagens, beneficiam-se desse tipo de arquitetura, uma vez que executam a mesma operação sobre vários dados distintos. Inicialmente, para utilizar as GPUs, era necessário o mapeamento dos dados utilizados para uma matriz, além de adequar as operações necessárias para o cálculo matricial. Nesse contexto, surgiram *frameworks* para abstrair esse tipo de limitação, como o CUDA (*Compute Unified Device Archicture*)(NVIDIA, 2007) e o OpenCL (MUNSHI, 2009).

2.1 Arquitetura básica

De acordo com a taxonomia de Flynn (DUNCAN, 1990), as GPUs podem ser classificadas como uma arquitetura *SIMD* (*Single Instruction - Multiple Data*), uma vez que executa um mesmo conjunto de instruções sobre uma grande quantidade de dados. Dado que os dados são independentes entre si, é possível a execução do conjunto de instruções simultaneamente. Por esse motivo, é necessário uma grande quantidade de núcleos de execução, denominados *processing cores* ou, simplesmente, *cores*. Na figura 2.1, é possível observar a arquitetura genérica de uma GPU. A unidade de controle (*Controller Unit*) controla a execução das operações pelas unidades de execução. Os dados de entrada e saída são colocadas na memória do dispositivo (*GPU Global Memory*).

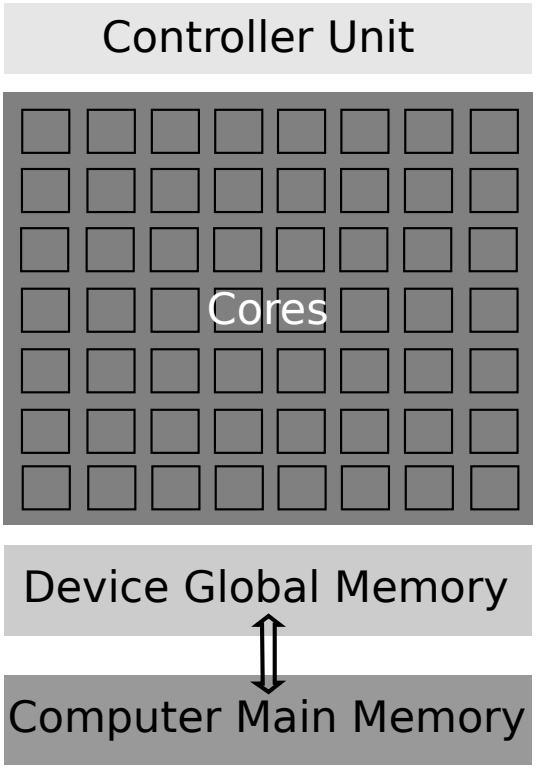


Figura 2.1: A arquitetura genérica de uma GPU

Uma vez que uma unidade de controle gerencia a execução de diversos *cores*, diferenças nos fluxos de execução das operações causam perdas de desempenho, já que a execução do caminho alternativo deve ser executado após o termino da computação do fluxo esperado. Além disso, é necessário controlar quais *cores* executam o caminho alternativo e quais, o caminho esperado.

A comunicação com o dispositivo é realizada através de sua memória global. As aplicações escrevem os operandos na memória global e os recuperam após o término das operações. As instruções a serem executadas também são escritas em um trecho da memória global.

2.2 CUDA

Nos últimos anos, dado o crescimento na capacidade de processamento das placas gráficas, que ultrapassou os *Teraflops* (*Tera floating point operations per second* - trilhões de operações de ponto flutuando por segundo), pôde-se observar um aumento na utilização desses processadores para a execução de aplicações genéricas. Tendo em vista esse cenário, a Nvidia, empresa especializada na fabricação de placas gráficas, criou o *CUDA*, com o intuito de facilitar o emprego de seus produtos em GPGPU. CUDA é um *framework* de desenvolvimento para aplicações genéricas e, por esse motivo, demandou uma revisão na arquitetura dos dispositivos, de modo a facilitar sua aplicação nessas tarefas. A utilização desse *framework* no desenvolvimento de aplicações abstrai vários detalhes de funcionamento das GPUs. Por exemplo, não é necessário o mapeamento dos dados para vértices ou fragmentos, como acontecia anteriormente (HARRIS et al., 2002).

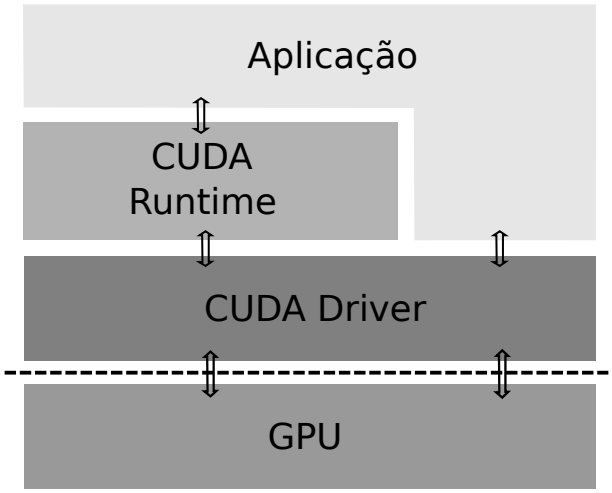


Figura 2.2: A estrutura das chamadas no *framework* CUDA

2.2.1 Organização de execução

O *framework* CUDA possibilita a utilização de GPGPU em aplicações escritas em C/C++, Fortran, OpenCL, DirectCompute, entre outras linguagens. Para isso, as instruções devem ser compiladas para uma GPU específica. Um programa executável na GPU é denominado *parallel kernel*, uma vez que é pode ser dividido vários fluxos de execução, denominados *threads*. Cada uma dessas estruturas é a menor conjunto de instruções de um *kernel* que realiza determinada operação. Cada uma das *threads* possui memória privada, registradores e um *program counter*, que gerencia o ponto de execução. As *threads* podem ser agrupadas em blocos. Um bloco é um conjunto de *threads* que pode cooperar entre si, através de sincronização ou memória compartilhada. Vale ressaltar que todas as *threads* têm acesso a memória global do dispositivo.

Para melhor explicar essa organização, será abordado uma aplicação paralela simples, como a multiplicação de matrizes. Cada um dos valores da matrizes resultantes é calculado independentemente dos outros. Por isso, cada *thread* calcula um valor da matriz resultante, multiplicando os operandos necessários. Essas *threads* podem ser agrupadas de maneira de acordo com o número de *cores* disponíveis no dispositivo, formando blocos. Esses blocos, serão agrupados para formar o *parallel kernel*.

2.2.2 Programação

Para a utilização do CUDA, são fornecidas duas APIs para a programação: *CUDA Runtime API* e *CUDA Driver API* (NVIDIA, 2010). A diferença entre essas APIs é o nível de abstração no controle da GPU. O *CUDA Runtime API* disponibiliza chamadas com maior nível de abstração, evitando que o usuário se preocupe com algumas configurações necessárias para a execução de sua aplicação. Já o *CUDA Driver API* disponibiliza chamadas que possibilitam um maior controle sobre a GPU. Porém, utilizando essa API, é necessário a realização das configurações de ambiente para a execução das aplicações, como a alocação da GPU para a aplicação e o carregamento do *kernel* para a memória do dispositivo. Como pode-se observar na Figura 2.2, o *CUDA Runtime* utiliza o *CUDA Driver* para acessar a GPU, realizando as chamadas de mais baixo nível pelo usuário.

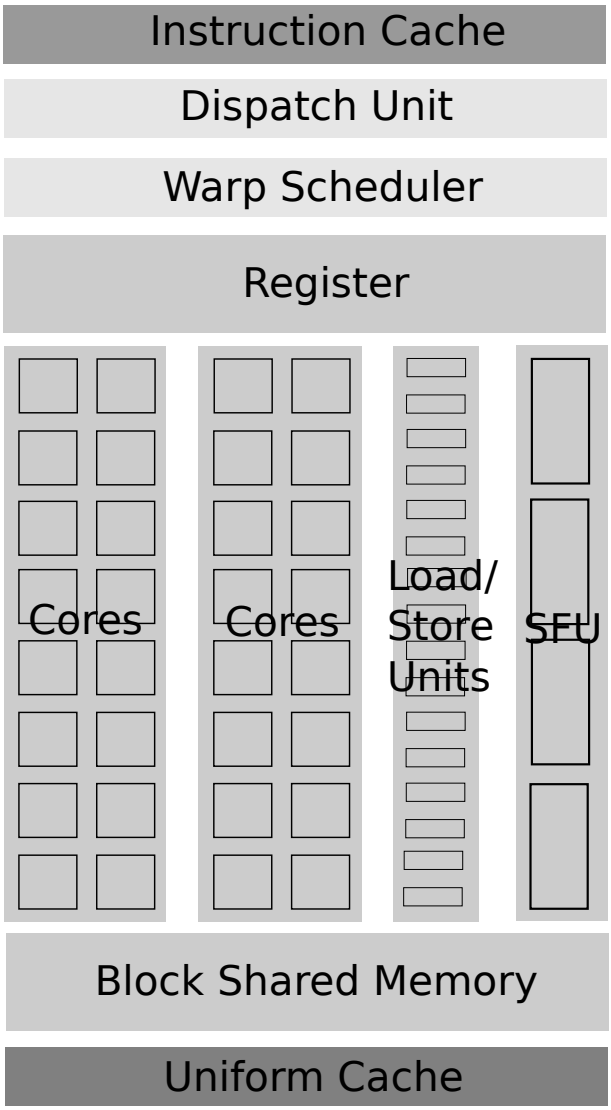


Figura 2.3: Esquemático da arquitetura de um *Streaming Multiprocessor*

2.2.3 Arquitetura

A arquitetura das placas gráficas compatíveis com o CUDA é baseado na utilização de multiprocessadores denominados *Streaming Multiprocessors* (SMs). Por serem modulares e escaláveis, o número desses multiprocessadores em cada modelo de GPU pode variar, sem ser necessárias grandes alterações no funcionamento das estruturas de controle. Como pode-se observar na figura 2.3, cada SM é composto por diversos *cores*, denominados *CUDA cores*, além de memória *cache* para instruções (*Instruction Cache*) e dados (*Uniform Cache*), banco de registradores, escalonadores e memória compartilhada. Pode-se observar ainda unidades para operações específicas, como unidades de carga/descarga de operandos (*Load/Store Units* - LDU) e unidades para funções especiais (*Special Function Units* - SFU).

É importante ressaltar que, como mostrado na arquitetura, um SM possui escalonadores (*Warp Scheduler*) e unidades de despacho (*Dispatch Units*). Os escalonadores são responsáveis por planejar a execução das *threads* pelos *cores* e a unidade de despacho têm o objetivo de enviar as tarefas para um grupo de unidades de execução. Essas unidades podem ser classificadas em três grupos: *Special Functions Units*, responsáveis pela exe-

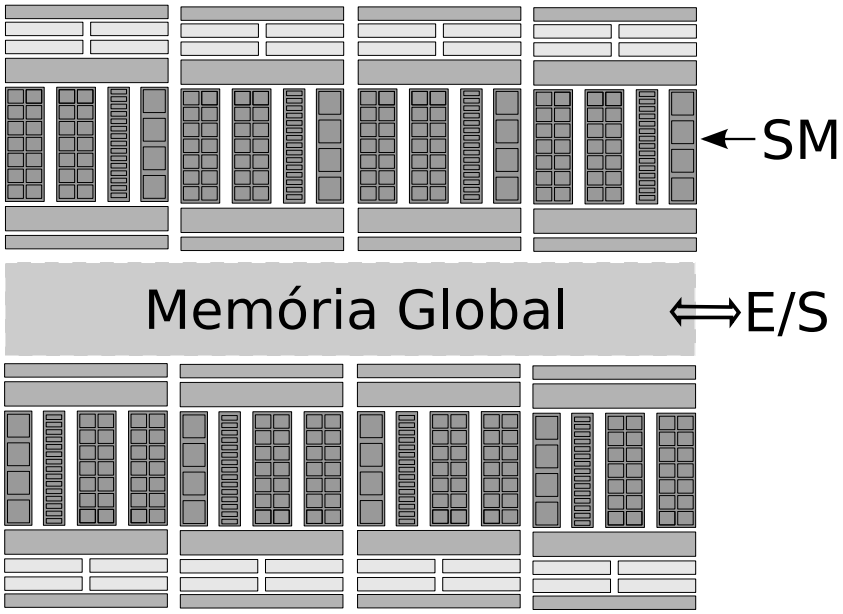


Figura 2.4: Esquemático da arquitetura de uma GPU compatível com CUDA

cução de funções matemáticas complexas; *Load/Store Units*, responsáveis pelo carregamento dos operandos da/para memória; e os *CUDA cores*, cujo objetivo é a execução de operações simples sobre números de ponto fixo e flutuante. Além disso, as unidades de execução podem se comunicar através da memória compartilhada (*Block Shared Memory*). Os registradores são utilizados pelas unidades de execução para armazenamento de dados durante sua execução. Ainda é possível observar uma cache de dados (*Uniform Cache*), utilizada para a guardar dados da memória global do dispositivo. Na figura 2.4, pode-se observar a arquitetura geral do dispositivo.

2.3 Considerações finais

Neste capítulo, foi estudada as unidades de processamento gráfico. Seu funcionamento é baseado na utilização de diversas unidades de execução, tornando sua programação complexa. Por esse motivo, foram desenvolvidos *frameworks*, como o CUDA, para facilitar o desenvolvimento de aplicações para esse dispositivo. O CUDA, além de uma ferramenta para desenvolvimento, é também uma arquitetura, baseada na utilização de *Streaming Multiprocessors*. Esses multiprocessadores disponibilizam diversas unidades de execução, possibilitando a execução simultânea das diversas *threads*, agrupadas em um *parallel kernel*, estrutura responsável pela execução da aplicação na GPU.

3 VIRTUALIZAÇÃO DE GPU

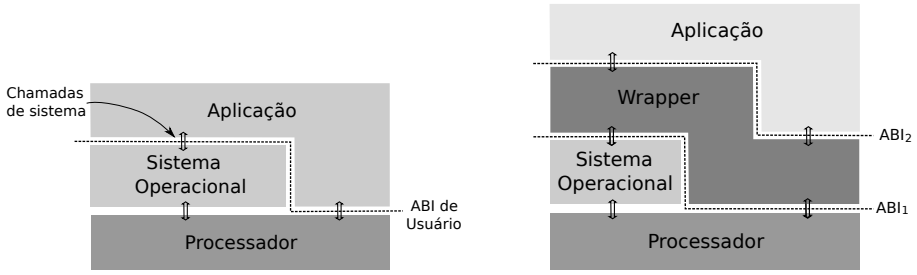
A virtualização consiste na emulação do comportamento dos dispositivos físicos presentes no sistema, escondendo suas características reais das aplicações em execução. Para isso, normalmente é criado uma camada intermediária de acesso ao dispositivo, denominada camada de virtualização (*Virtualization Layer*), que funciona como um filtro de acesso, onde o dispositivo deve, obrigatoriamente, ser utilizado através dessa estrutura. Um exemplo disso são as máquinas virtuais (*Virtual Machines* - VMs), que emulam o comportamento dos dispositivos físicos, possibilitando a execução de sistema operacional (*Operating System* - OS).

3.1 Conceitos básicos de virtualização

Com o aumento na capacidade dos dispositivos, surgiu a possibilidade de execução de mais uma aplicação sobre um mesmo *hardware* físico. Dessa forma, os aplicativos alocariam recursos de *hardware* para executar suas tarefas. Todavia, era necessária uma entidade para uma camada de *software*, cuja função é gerenciar a alocação de recursos entre as aplicações, possibilitando o compartilhamento. Essa camada de *software* é denominada Sistema Operacional (*Operating System* - OS).

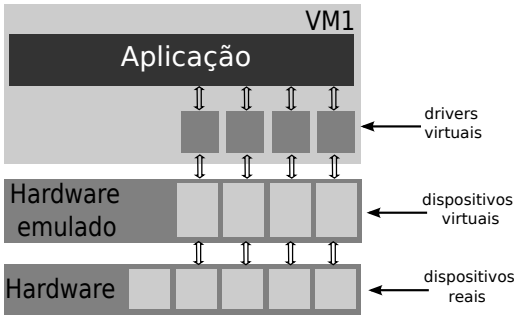
Utilizando essa camada, várias aplicações podem compartilhar os dispositivos físicos da máquina, utilizando, para isso, técnicas como o *round-robin*, *FIFO*, que estão fora do escopo deste trabalho. Devido a esse compartilhamento de recursos, a aplicação não pode acessar diretamente o dispositivo, uma vez que ele pode estar indisponível ou já alocado no momento. Por esse motivo, as operações sobre os dispositivos deveriam ser requeridas ao OS, que efetuaria as operações necessárias sobre o *hardware* físico. Para possibilitar essa comunicação, o sistema operacional disponibiliza um conjunto de chamadas de sistemas (*system call*).

Por esse motivo, as instruções dos processadores atuais requerem uma determinada permissão para sua execução. Pode-se observar, na figura 3.1(a), que algumas instruções da aplicação devem ser executadas através de chamadas de sistema. Por esse motivo, as aplicações podem executar um conjunto de instruções e chamadas de sistemas. Esse conjunto é denominado ABI (*Application Basic Interface*). Para emular o funcionamento de outros processadores, é possível disponibilizar uma ABI diferente, através da utilização de um *wrapper*, como pode-se observar na figura 3.1(b). Esse *wrapper* seria responsável pela transformação de instruções de uma ABI, no caso a *AB2*, para outra ABI, ABI1.

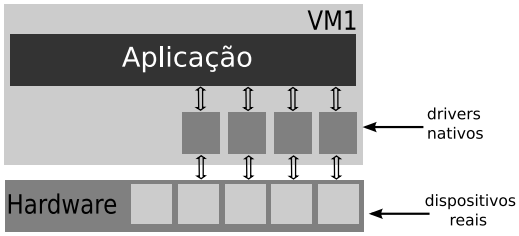


(a) Conjunto de instruções disponibilizadas à aplicação

(b) Utilização de um *wrapper* para emular ABIs



(c) Emular o dispositivo



(d) Possibilitar o acesso direto

Figura 3.1: Abordagens para a virtualização de dispositivos

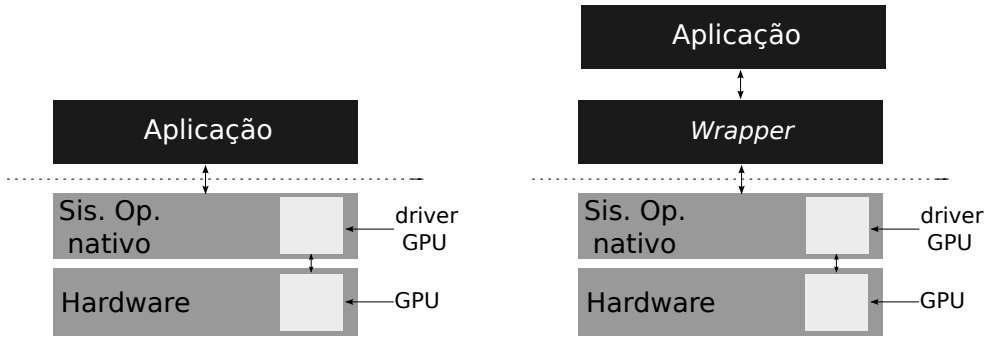
3.2 Virtualização e E/S

As máquinas reais são formadas por diversos dispositivos de entrada/saída (E/S), como discos rígidos, placas de rede, teclado, entre outros. Esses dispositivos são imprescindíveis para o funcionamento do sistema, uma vez que possibilitam a comunicação do processador com outros computadores e/ou usuários. Por isso, é necessário que as máquinas virtuais possibilitem a utilização desses dispositivos. Todavia, a abordagem apresentada anteriormente, não é aplicável para esse tipo de dispositivo. Para disponibilizam esses dispositivos, as VMs podem emulá-los ou possibilitar o acesso direto.

Ao emular um dispositivo, as máquinas virtuais disponibilizam um dispositivo *virtual* ao sistema operacional convidado (*Guest OS*), ou seja, aquele executando na VM. Aplicações executando na VM acessam esses dispositivos e, as operações executadas, são repassadas aos dispositivos físicos. Essa abordagem pode ser melhor observada na figura 3.1(c). A aplicação utiliza *drivers virtuais* para o acesso ao *dispositivo virtual*. Todas as operações executadas são repassadas para o *hardware físico* da máquina.

3.3 Virtualização de GPU - Proposta

As placas de processamento gráfico funcionam como um dispositivo de E/S, uma vez que se comunicam através de um barramento, possuem memória interna, entre outras características. Quando uma aplicação utiliza esse dispositivo para a realização de computação genérica, o *driver* disponível no sistema operacional nativo é acessado diretamente pela entidade requisitante, como mostrado na figura 3.2(a). É possível a utilização de uma camada de consistência entre a aplicação e o *driver* de dispositivo. Esse tipo de acesso é mostrado na figura 3.2(b), onde essa segunda camada de acesso é denominada de *wrapper*.



(a) Acesso direto ao dispositivo (b) Acesso indireto ao dispositivo

Figura 3.2: Esquemático dos tipos de acesso a GPU

As funcionalidades do *wrapper* podem variar desde um simples redirecionamento de tarefas até um complexo gerenciamento do dispositivo. Utilizando esse método, pode-se ainda esconder as características físicas do sistema. Por exemplo, caso a máquina disponha de *n* GPUs. As aplicações não precisam saber quantas e quais as GPUs disponíveis, é necessário apenas o envio das requisições para o *wrapper*, que se engarrega de reencaminhá-las para um recurso disponível. Uma outra aplicação seria a utilização de um *cluster* de GPUs. O *wrapper* poderia alocar *remotamente* a GPU no *cluster*, possibilitando que máquinas sem GPU executem aplicações que necessitem desse tipo de recurso. Vale ressaltar que durante o desenvolvimento desse trabalho, essa abordagem teve de ser implementada devido a escassez de recursos, mostrando que esse tipo de aplicação é plausível de estudo.

O *wrapper* pode ser aplicado para disponibilizar o acesso às GPUs nas máquinas virtuais. Essa camada pode encaminhar as requisições das aplicações executando na VM para o *driver* nativo, possibilitando a execução de aplicações de GPGPU.

No contexto de virtualização de GPUs, esse *wrapper* pode ser implementado de duas maneiras: emulando um dispositivo genérico ou encaminhando as funções para uma API nativa. Ao emular um dispositivo genérico, esse *wrapper* deve converter as tarefas para a execução na GPU, o que pode ser muito oneroso. Todavia, essa generalização possibilita a rápida migração de máquinas virtuais entre os servidores, dado que as APIs disponibilizadas as VMs não mudam. Por outro lado, ao disponibilizar uma API equivalente a API nativa, espera-se um ganho de desempenho, mas pode-se perder a portabilidade das VMs.

Esse *wrapper* também pode ser utilizado para virtualizar um *framework* de desenvolvimento de GPGPU, como o CUDA e o OpenCL. O wrapper disponibilizaria a API do *framework* e não a do *driver*. Desse modo, o *wrapper* só seria alterado em caso do alteração no *framework*. Alterações no funcionamento e no *driver* do dispositivo não implicariam em alterações.

3.4 Considerações finais

Neste capítulo, foi abordado o conceito de virtualização, focando nas máquinas virtuais. A utilização de VMs facilita o gerenciamento dos diversos serviços em execução em uma infraestrutura, uma vez que são necessárias menos máquinas físicas. Ainda sim, uma máquina virtual necessita de dispositivos de E/S, possibilitando a comunicação com o usuário e outras máquinas. Dado que as máquinas virtuais executam sobre uma camada de *software*, com ou sem a presença de um OS nativo, é necessário garantir o acesso da

VM aos dispositivos físicos, que podem ser disponibilizados através de uma camada de virtualização.

4 IMPLEMENTAÇÃO

Tendo em vista o objetivo deste trabalho, que é possibilitar a utilização de GPGPU em aplicações executando em máquinas virtuais, e os conceitos mostrados nos capítulos anteriores, é possível descrever a implementação da solução proposta, bem como as decisões tomadas durante o desenvolvimento. Também é possível a definição de uma arquitetura básica de funcionamento, mostrando a possibilidade de sua aplicação em outros contextos.

4.1 Requisitos e decisões

Para implementar a solução descrita neste trabalho, foi necessário alterar uma ferramenta de virtualização de forma a possibilitar o acesso da máquina virtual à GPU. Primeiramente, a escolha da ferramenta de virtualização foi essencial, dado que a implementação seria baseada nessa ferramenta. Outra escolha importante era o *framework* de desenvolvimento de GPGPU. Tendo em vista esse cenário, foram levantados alguns requisitos para a escolha das ferramentas:

- Fácil utilização: As ferramentas deveriam ser de fácil instalação e configuração, de modo a possibilitar que usuários leigos pudessem utilizá-la.
- Aceitação de mercado: As ferramentas deveriam ser conhecidas e utilizadas no mercado. A aplicação da solução em ferramentas desconhecidas ou pouco aplicáveis ao contexto do mercado implica em pouco impacto do estudo na área de pesquisa.
- Multi-plataforma: As ferramentas deveriam suportar diferentes OS, dado a diversidade encontrada nas infraestruturas atualmente.
- Documentação: A documentação é essencial para o entendimento sobre o funcionamento da ferramenta, suas limitações e vantagens.

Além desses requisitos, a ferramenta de virtualização deveria ser disponibilizada através da licença GPL2 (LICENSE, 2006), possibilitando a execução das modificações necessárias. Analisando as ferramentas disponíveis, o *VirtualBox OSE* foi escolhido como ferramenta de virtualização. Essa ferramenta possui uma documentação abrangente e sua aplicação em *desktops* é notável, dado a intensidade de discussões nos fóruns. Além disso, sua instalação e configuração são extremamente simples.

Para o desenvolvimento de aplicações de GPGPU, foi escolhido o *framework CUDA*, devido a disponibilidade de dispositivos, sua aceitação no mercado atualmente e a maturidade da ferramenta. A documentação é bem completa e o desenvolvimento de aplicações

é bem simples, como podemos observar no capítulo 2, onde esse *framework* foi descrito mais profundamente.

Após a escolha das ferramentas-base para a implementação, foram levantados alguns requisitos para o funcionamento da solução:

- Compatibilidade: As aplicações desenvolvidas para a execução no OS nativo deveriam ser compiladas e executadas sem alteração no código-fonte.
- Facilidade de utilização: Não deveria impor grandes alterações na configuração das ferramentas envolvidas e na máquina do usuário.
- Multi-plataforma: A solução deveria ser aplicável em diferentes OS.
- Estabilidade: A solução deveria utilizar serviços estáveis das ferramentas, de modo que alterações na versão da ferramenta não comprometessem sua aplicação.

Tendo em vista esses requisitos, a solução foi descrita e sua arquitetura, definida. No decorrer deste capítulo, descrevemos melhor as soluções utilizadas para a implementação da solução e seu impacto no desenvolvimento.

4.2 Arquitetura modular do VirtualBox

O *VirtualBox OSE* é um VMM *hosted* com suporte a virtualização total, mantida pela Oracle. Por esse motivo, o núcleo da ferramenta é um módulo denominado *hypervisor*. Esse módulo é responsável pelas atividades requeridas pela virtualização total, como emular os dispositivos físicos, teste e desvio das instruções executadas, entre outras coisas, além de gerenciar a comunicação entre o OS nativo e o OS convidado.

Em uma ferramenta de virtualização, a comunicação entre o OS nativo e o OS convidado é extremamente importante, uma vez que os dispositivos de entrada e saída, como teclado e *mouse*, estão conectados ao OS nativo. Dessa forma, é imprescindível que o OS convidado acesse as informações enviadas por esses dispositivos. Além disso, existem diversas aplicações que utilizam essa comunicação em sua execução. Por esse motivo, a VirtualBox disponibiliza formas de comunicação entre o OS nativo e o OS convidado, podendo destacar o HGSMI e o HGCM, descritos mais profundamente no decorrer deste trabalho.

4.2.1 HGSMI - Host-Guest Shared Memory Interface

O HGSMI é uma interface para o compartilhamento de memória entre o OS convidado e o OS hospedeiro, possibilitando a comunicação entre aplicações executando nesses dois ambientes. O HGSMI oferece melhor desempenho, porém a complexidade de implementação é elevado, quanto comparado com o HGCM. Além disso, a documentação disponível sobre esse método é escassa.

4.2.2 HGCM - Host-Guest Communication Manager

O HGCM é um método de comunicação que possibilita o compartilhamento de bibliotecas, possibilitando que aplicações executando na VM utilizem funções implementadas no OS nativo. Essa biblioteca compartilhada é implementada como um serviço disponibilizado pelo OS nativo. Sendo assim, o OS convidado requisita uma determinada tarefa para esse serviço.

Nome do campo	Descrição
size	Tamanho da requisição
version	Versão do cabeçalho
type	Tipo da requisição
rc	código de retorno HGCM, que será alterado pelo dispositivo virtual
flags	*flags* para controle do dispositivo virtual
result	resultado da chamada de função, alterado pelo dispositivo virtual
params	parâmetros da requisição do serviço

Tabela 4.1: Estrutura de um cabeçalho de requisição HGCM

O primeiro passo dessa comunicação é a conexão da aplicação requisitante com o serviço disponibilizado. Vale ressaltar que essa aplicação está em execução dentro da máquina virtual. Ao realizar essa conexão, o serviço HGCM identifica o requisitante, disponibilizando-o um identificador. Além disso, essa conexão possibilita verificar a disponibilidade do serviço. Uma vez conectado, o cliente constrói, na sua memória física, a requisição da tarefa, escrevendo as informações necessárias, descritas na tabela 4.1. A requisição é então submetida para o OS nativo.

Quando as tarefas solicitadas estão concluídas, a requisição é marcada como completa e uma interrupção é lançada na VM. Dessa forma, a aplicação requisitante consulta os dados retornados. Quando não existem mais tarefas, a aplicação requisita sua desconexão do serviço. Nesse momento, os dados de identificação do cliente são descartados. Vale ressaltar que a comunicação é sempre iniciada pelo cliente na máquina virtual.

4.3 Interface CUDA driver

Como descrito no capítulo 2, o *framework* CUDA possibilita a utilização da GPU para aplicações genéricas. Esse *framework* disponibiliza duas APIs para o desenvolvimento de aplicações: o *CUDA Runtime* e o *CUDA Driver*. O *CUDA driver* oferece uma API de mais baixo nível de abstração. Essa API é utilizada pelo *CUDA Runtime* para acessar a GPU.

Vale ressaltar que o *CUDA driver*, apesar do nome, é uma biblioteca de funções disponibilizada pela Nvidia. Ela utiliza o *driver* da GPU para acessar o dispositivo e enviar os dados necessários. A aplicação utiliza o *CUDA driver*, que, por sua vez, utiliza o *driver* da GPU do sistema operacional, como mostrado na figura 4.1.

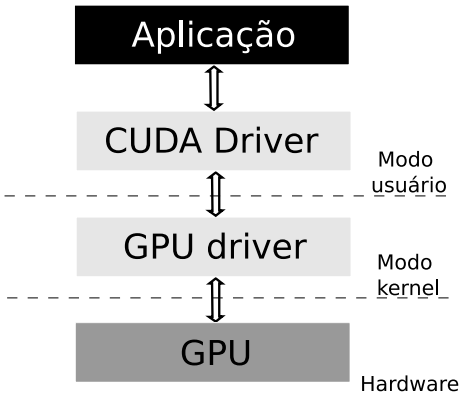


Figura 4.1: Esquemático de organização do acesso à GPU

4.4 VirtCUDA

De modo a possibilitar a utilização de GPUs nas máquinas virtuais, é necessário uma camada de virtualização que disponibilize o acesso à GPU. Como discutido na seção 3.3, essa camada de virtualização pode disponibilizar a API do *framework* de GPGPU ou a do *driver* do dispositivo. Já que o foco deste trabalho é possibilitar a aplicação de GPGPU em máquinas virtuais, a disponibilização da API do *framework* foi adotada para a implementação, devido ao contexto atual do mercado, com o lançamento de novas tecnologias no âmbito de processadores gráficos, e a facilidade de compartilhamento de bibliotecas no *VirtualBox OSE*, através do HGCM.

Uma vez que o *framework CUDA* utiliza o *CUDA Driver* para acesso a GPU, seu compartilhamento entre o OS nativo e o OS convidado possibilita a utilização de todo o *framework* no ambiente virtualizado. Por isso, qualquer aplicação desenvolvida utilizando o *CUDA* pode ser executada dentro da máquina virtual.

Para disponibilizar o acesso ao *CUDA driver*, é possível a construção de uma biblioteca que funcione como um cliente HGCM. Esse cliente envia requisições para o serviço HGCM, que as reencaminha para o *CUDA driver* no OS nativo. O *CUDA driver* executa a tarefa requisitada e envia o resultado para o serviço HGCM, que o reencaminha para o cliente HGCM. Dessa forma, a GPU pode ser acessada através dessa estrutura, como mostrado na figura 4.2.

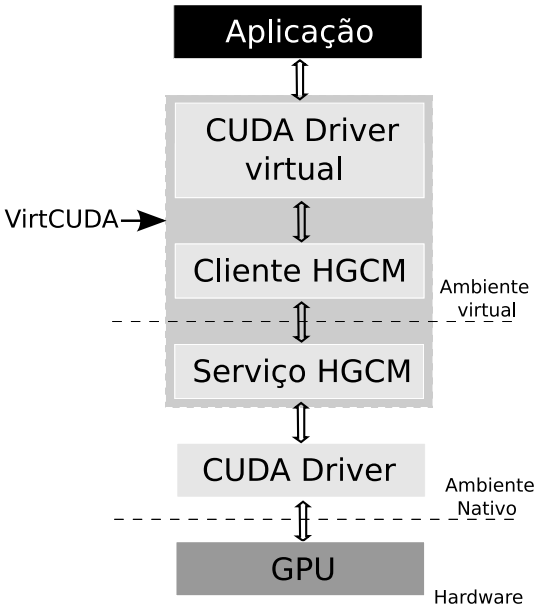


Figura 4.2: Organização do módulo VirtCUDA

Para a implementação dessa funcionalidade, foi necessário a disponibilização de um *CUDA Driver virtual*, que trataria as requisições como o driver original. Por esse motivo, é necessária a implementação de todas as funções disponíveis no *CUDA driver* original. A API do CUDA disponibiliza um grande número de funções. Para fins de teste da solução, apenas as mais utilizadas foram implementadas, que estão devidamente descritas no anexo referido.

Vale ressaltar que a arquitetura descrita pode ser utilizada com outros *frameworks* de GPGPU, como o *OpenCL*. Para isso, é necessário seguir a mesma arquitetura descrita nessa seção e disponibilizar a API do *framework* em questão, ao invés, da API do *CUDA driver*.

4.5 Desafios

Apesar da arquitetura de implementação da solução parecer extremamente simples, com módulos simples e que realizam funções bem definidas no contexto da aplicação, é possível a definição de diversos desafios de implementação. Esses desafios ocorrem como consequência de diferenças entre as GPUs e as CPUs, do gerenciamento de memória do dispositivo e o compartilhamento de memória entre a VM e o OS nativo. Os problemas encontrados serão descritos detalhadamente no decorrer deste capítulo.

4.5.1 Salvamento de contexto

Uma vez que as aplicações executando dentro da VM podem acessar os dispositivos reais e é possível a execução de várias VMs simultaneamente, é necessário compartilhar esses recursos entre as máquinas virtuais, ou seja, é necessário escalonar a alocação do dispositivo pelas aplicações. Durante o compartilhamento, é necessário o salvamento completo do contexto da VM, incluindo a memória utilizada, dispositivos em atividade, entre outras coisas. Normalmente, para salvar o contexto de um dispositivo, são necessários as ações descritas a seguir.

1. Parar a execução das operações sobre o dispositivo ou esperar que as operações sejam concluídas
2. Salvamento do ponto de execução da tarefa do dispositivo
3. Salvamento dos dados nos registradores e memória do dispositivo
4. Desalocar o dispositivo real da VM

Esses passos são aplicáveis para o salvamento de contexto de qualquer tipo de recurso, incluindo a CPU. Nesse contexto, um dos problemas encontrados para aplicação dessa técnica em GPUs é justamente falta de instruções para o salvamento de contexto. Ou seja, não é possível salvar o ponto de parada exato das *threads* executando no dispositivo, dado que essas estruturas são independentes e podem não estar totalmente sincronizadas. Essa limitação inviabiliza o salvamento, uma vez que será necessário o salvamento do ponto de execução de *todas as threads em execução*. Todavia, essa tarefa normalmente consume mais recursos do que esperar pelo término da execução da aplicação na GPU.

Uma possível solução para esse problema é esperar o fim da computação e, após esse momento, salvar a memória do dispositivo. Vale ressaltar que, para utilizar a GPU, a aplicação envia dados para a memória do dispositivo, espera pela sinalização de término da execução e recupera os dados da memória da GPU. Desse modo, para evitar perdas de desempenho, as aplicações devem ser paradas no momento em que recuperam totalmente a memória da GPU, evitando a necessidade de realizar mais uma cópia de memória.

É importante ressaltar que essa funcionalidade ainda é uma questão em aberto, devido a dificuldades técnicas enfrentadas. A implementação dessa funcionalidade serve como trabalho futuro para este trabalho.

4.5.2 cudaHostAlloc

O *framework* CUDA disponibiliza diversas funções para o gerenciamento da GPU. Para facilitar o desenvolvimento, são disponibilizadas funções que manipulam alguns recursos da própria máquina, como a memória principal. Dessa forma, é possível utilizar a memória principal de modo síncrono com a GPU, facilitando o controle sobre os

dados e a utilização da GPU pela aplicação. Dentre essas funções, pode-se destacar a *cudaHostAlloc*. O objetivo dessa função é alocar uma porção de memória fixa (*pinned memory*), ou seja, uma porção de memória não-paginável na memória principal, facilitando a comunicação com a GPU.

O principal problema na utilização dessa função é criado pelo encapsulamento da VM. O espaço de endereçamento na VM é totalmente independente do OS nativo. Além disso, a aplicações na VM não possuem direitos de acesso à memória do OS nativo. Por esse motivo, se uma porção de memória fixa é alocada no espaço de endereçamento do OS nativo, a VM não tem permissão para alterar os dados. Por outro lado, se alocarmos uma porção de memória fixa na VM, a GPU não pode acessá-la, uma vez que o espaço de endereçamento é totalmente independente na VM.

Desse modo, para a utilização do *cudaHostAlloc*, é necessário uma sincronização de memória entre a porção fixa no OS nativo e a memória alocada na VM. Essa sincronização deve ser feita imediatamente antes do início da computação pela GPU e imediatamente depois do término da execução. Todavia, essa sincronização acarretaria em graves perdas de desempenho, dada a necessidade de várias cópias de memória. Uma abordagem alternativa seria compartilhar esse trecho de memória com a VM.

Vale ressaltar que a implementação dessa funcionalidade está sendo estudada. Por esse motivo, serve como trabalho futuro.

4.5.3 Dificuldades no Windows

De acordo com os requisitos descritos na seção 4.1, a solução deve ser aplicável em diferentes OS. Para o desenvolvimento, foram escolhidas duas plataformas: *Linux* e *Windows*, que representam a grande maioria de sistemas instalados nas infraestruturas atualmente.

Durante as fases iniciais do desenvolvimento, foi necessário a compilação e executação da ferramenta *VirtualBox OSE*, de modo a avaliar seu funcionamento e iniciar o desenvolvimento. Nos sistemas baseados em *Linux*, o processo de configuração e compilação foi simples, devido aos vários tutoriais disponibilizados na rede.

Todavia, durante a configuração e compilação da ferramenta para *Windows*, foram encontrados diversas dificuldades, causados principalmente pelo ciclo de vida das ferramentas da *Microsoft*. Várias das ferramentas necessárias foram descontinuadas e, por esse motivo, não foi possível encontrá-las para *download*. Uma vez que o *VirtualBox OSE* ainda não apresentava compatibilidade com as novas ferramentas, a compilação da ferramenta e possível aplicação da solução em sistemas *Windows* foi postergada.

4.6 Considerações finais

No presente capítulo, foi descrita a arquitetura da solução proposta, bem como os serviços e ferramentas que oferecem suporte a implementação. O serviço HGCM possibilita o compartilhamento de bibliotecas entre a VM e o OS nativo. Utilizando esse serviço, é possível compartilhar o *CUDA driver* com a VM. Dado que o *CUDA driver* é a base de funcionamento do *framework* CUDA, também é possivel a utilização de aplicações desenvolvidas utilizando a API do *CUDA Runtime*. Por esse motivo, a maior parte das aplicações pode ser executada dentro da VM.

5 AVALIAÇÃO

A utilização de máquinas virtuais apresenta várias vantagens, como uma melhor alocação dos recursos e um maior isolamento dos ambientes de execução de cada aplicação. Todavia, ao disponibilizar um recurso para a VM, é necessário adicionar uma camada de virtualização, o que gera uma série de *overheads* na utilização do dispositivo.

Desse modo, a virtualização da GPU adiciona um custo a sua utilização, gerado pela camada de virtualização. Todavia, os *overheads* gerados não devem aumentar demasiadamente o tempo de execução da aplicação. Tendo em vista esse cenário, a avaliação da solução proposta neste trabalho é extremamente importante. Esse estudo possibilita desvendar os custos envolvidos na virtualização do dispositivo.

5.1 Metodologia

Para a avaliação da solução proposta, são executados aplicações de modo a avaliar alguns aspectos do funcionamento do sistema. Desse modo, é possível determinar seu comportamento durante sua utilização, além de medir os custos inseridos pela camada de virtualização.

Uma vez que o objetivo deste trabalho é disponibilizar a GPU para a VM, os testes serão executados na máquina virtual e na máquina física. Durante essas execuções, foram feitas medidas de tempo, de modo a avaliar o *overhead* gerado. Além disso, alguns custos variáveis, como transferência de memória, serão avaliados, possibilitando uma melhor avaliação do comportamento do sistema.

5.1.1 Plataforma

A execução das aplicações é extremamente dependente dos recursos disponíveis para sua execução. Uma vez que as aplicações são executadas na máquina física e na VM, os dispositivos disponíveis nesses dois sistemas influenciam a execução. Na tabela 5.1, pode-se observar os recursos disponíveis em cada um dos sistemas citados. Por sua vez, na tabela 5.2, as especificações da GPU utilizada.

5.1.2 Benchmarks utilizados

Para avaliar a solução, foram utilizados aplicações de terceiros. Cada uma das aplicações foi desenvolvida por um programador, o que implica em estruturas e algoritmos totalmente diferentes. Dessa forma, foi possível observar o comportamento da execução de aplicações genéricas, algo próximo ao encontrado nos ambientes produtivos das organizações.

As aplicações de teste deveriam ser extremamente paralelizáveis, possibilitando uti-

Recurso	Máq Física	VM
Processador	Intel Core 2 Duo P7450 2.13 GHz 3MB *cache L2*	
Número de cores	2	1
Memória física	4GB DDR3 1066 MHz	512MB DDR3 1066MHz
OS	Ubuntu 10.04 LTS	Ubuntu 10.04 LTS
kernel	2.6.32-25-generic	2.6.32-21-generic

Tabela 5.1: Recursos da máquina física

Item	Descrição
Nome	GeForce G210M
Memória Principal	256MB GDDR3
CUDA cores	16
Frequência de *clock*	1500MHz
Barramento	PCI-E 2.0

Tabela 5.2: Especificações da GPU

lizar o máximo de recursos disponíveis. Por esse motivo, o campo de processamento de imagens ofereceu ótimos exemplos para estudo, dado a existência de várias aplicações disponíveis, possibilitando a avaliação da solução com aplicações desenvolvidas por programadores alheios a este trabalho. Além das análises sobre as transferências de memória, pode-se destacar os seguintes testes:

- Filtragem de imagens: A aplicação de filtros em imagens baseia na multiplicação de matrizes. Essa aplicação foi desenvolvida pelos autores deste trabalho.
- Convolução de matrizes: Operação aplicada para melhorar a qualidade de imagens. Essa aplicação foi baseada nos exemplos disponibilizados pela Nvidia para testes do *framework* CUDA.
- Histograma: Essa operação é utilizada para contabilizar a frequência de cores em uma imagem. A aplicação foi desenvolvida por um colaborador francês do projeto.

É importante ressaltar que todos os testes foram executados 100 vezes, de modo a obter dados como a média harmônica, desvio padrão, variância, entre outros. Todavia, o desvio padrão e a variância medidos foram mínimos, em torno de $30\mu s$, e serão desprezados no decorrer da avaliação.

5.2 Resultados

Nesta seção, são apresentados os resultados obtidos com os testes aplicados, além de uma análise sobre os dados metidos. Dessa forma, é possível avaliar o funcionamento da ferramenta.

5.2.1 Transferências de memória

Para avaliar o custo gerado sobre a transferência de memória, foi criada uma aplicação, cujo objetivo era alocar um trecho de memória no dispositivo, transferir dados para esse trecho e recuperar esses dados. Dessa forma, é possível medir o tempo da transferência

da memória principal para o dispositivo e no sentido inverso. Os tempos de transferência nos dois sentidos foram extremamente parecidos e, por esse motivo, será abordado uma média entre esses tempos. Por esse motivo, no gráfico 5.1, pode-se observar o tempo de transferência relativo ao tamanho do trecho transferido. Vale ressaltar que, pelo funcionamento do HGCM, a memória não precisa ser copiada para o OS nativo, uma vez que esse sistema já possui acesso a esse trecho de memória. Por esse motivo, o custo adicionado à transferência de memória é gerado exclusivamente pela camada de virtualização. Esse custo, como pode-se observar no gráfico, é 0.4ms, 100 vezes maior do que o tempo de transferência de memória do sistema físico.

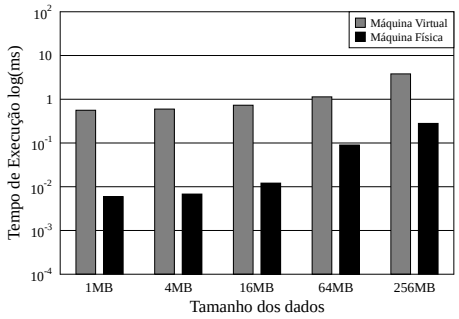


Figura 5.1: Tempo de transferências de dados para/da GPU

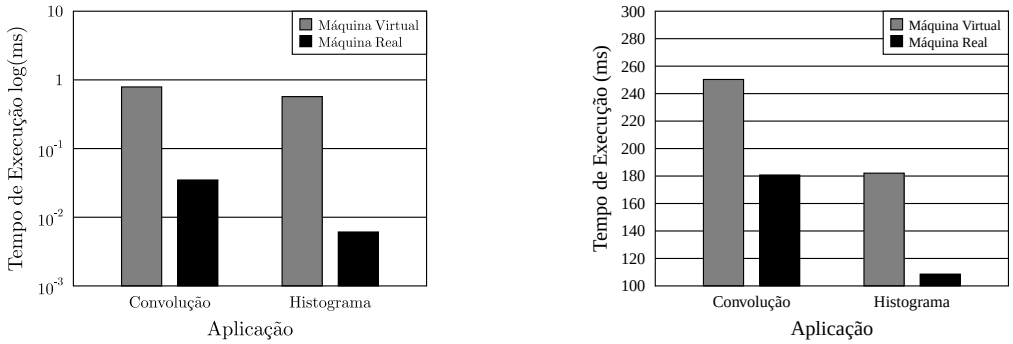
5.2.2 Convolução e Histograma

A convolução é aplicada para melhorar a qualidade de imagens e consiste em multiplicações sequenciais de matrizes. Dessa forma, é possível paralelizar essas multiplicações, de forma a utilizar todo o poder computacional para operações de ponto flutuante da GPU. Vale ressaltar que essa aplicação é estática, ou seja, o tamanho dos dados envolvidos não é controlado por parâmetros de execução da aplicação. Por isso, não foi possível variar o tamanho dos dados de entrada para uma melhor análise do comportamento do sistema.

Por sua vez, a determinação de um histograma permite averiguar a frequência de utilização do espectro de cores em uma imagem. Para determinar o histograma é necessário a classificação de todos os pixels formadores da imagem, agrupando-os. Dessa forma, pode-se averiguar a proporção de cada cor, possibilitando calcular o gasto de impressão, enérgico na visualização da imagem, entre outras coisas.

No gráfico 5.2(a), pode-se observar a diferença no tempo de execução do *kernel* das duas aplicações, ou seja, o tempo de processamento na GPU, dessas operações quando requisitadas pelo sistema nativo e pelo sistema virtualizado. O custo adicionado pela virtualização está em torno de 0.4ms. Quando comparado ao tempo de execução das operações na GPU, esse custo é bem alto, uma vez que a GPU é bastante rápida na execução de suas tarefas. Por esse motivo, o gráfico referenciado está em escala logaritmica, possibilitando a visualização dos tempos.

Todavia, na figura 5.2(b), pode-se observar a diferença na execução de toda a aplicação, o que inclui as transferências de memória, o processamento na CPU, a escrita e leitura nos *buffers* de OS. Desse modo, o custo medido não é tão alto, quando comparado ao custo de execução total da aplicação, dado que, nesse exemplo, o gráfico está em escala linear. Dessa forma, o custo do gerenciamento do HGCM é *diluído* no tempo de execução das outras operações, não afetando demasiadamente a execução da aplicação.



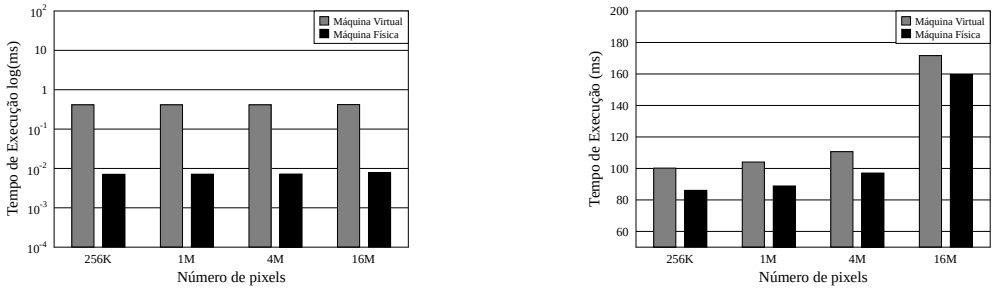
(a) Tempo de execução do *kernel* (b) Tempo total de execução

Figura 5.2: Tempo de execução das aplicações na GPU

5.2.3 Filtragem

A filtragem é aplicada para a remoção de ruídos na captação de imagens e consiste na aplicação de uma máscara sobre a imagem. Normalmente, essa máscara é aplicada em todos os pixels independentemente, gerando um alto grau de paralelismo nesse tipo de aplicação. Para aplicar uma determinada máscara sobre um *pixel*, é necessário uma multiplicação de matrizes, onde uma matriz é a mascara propriamente dita e a outra matriz é uma parte da imagem, onde o *pixel* alvo se encontra. Para avaliar o desempenho da GPU, cada um dos *pixels* formadores da imagem foi transformado em um número de ponto flutuante.

Essa aplicação foi desenvolvida pelos autores do trabalho, o que possibilitou a análise do comportamente da solução frente a variação no tamanho dos dados de entrada e saída. Na figura 5.3(a), pode-se observar o tempo de execução do *kernel* da aplicação em função do número de *pixels* da imagem a ser filtrada. O custo da virtualização é 0.4ms e se mostrou constante durante a variação nos dados de entrada. Já que o tempo de execução do *kernel* na GPU é pequeno quando comparado ao custo da virtualização, o gráfico 5.3(a) está em escala logarítmica. Desse modo, a virtualização adiciona um custo de 2 ordens de grandeza no acesso a GPU, quando comparado ao tempo de acesso no sistema físico.



(a) Tempo de execução do *kernel* (b) Tempo total de execução

Figura 5.3: Tempo de execução da aplicação de filtragem

Já na figura 5.3(b), pode-se observar o tempo total de execução da aplicação, o que inclui *todas as operações necessárias* para a execução da aplicação. A virtualização não

adicionou grandes custos, comparado com a execução no OS nativo, dado que os custos gerados pelo HGCM foram incorporados nas outras operações do sistema.

5.3 Considerações finais

Neste capítulo foi abordado a avaliação da solução proposta, bem como metodologia utilizada para avaliação. Foi possível observar um custo constante entre 0.4ms e 0.5ms. Esse tempo é gerado pelo funcionamento do HGCM. Após a análise de performance, não foi possível diminuir esse custo de utilização antes da escrita desse trabalho. Todavia, ao analisar o tempo total de execução das aplicações, foi constatado que os custos da virtualização são incorporados em outras operações do sistema. Dessa forma, o custo observado na execução de toda a aplicação foi pequeno. Além disso, a utilização das GPU em VMs torna o sistema mais confiável, dado algumas vantagens das máquinas virtuais descritas nos capítulos anteriores.

6 CONCLUSÃO

Tendo em vista o cenário descrito nos capítulos anteriores, o objetivo deste trabalho é propor um arquitetura genérica para a virtualização da GPU, possibilitando sua utilização por aplicações nas máquinas virtuais, bem como uma forma de implementação em uma ferramenta de virtualização, denominada *VirtualBox*. Além disso, foram executados testes de modo a avaliar o comportamento da solução, possibilitando determinar os custos de sua utilização. O custo medido foi alto, algo em torno de 0.4ms, quando comparado com a execução das operações na GPU. Todavia, a virtualização não alterou de maneira significativa o tempo total de execução da aplicação. Esses custos são observados normalmente na virtualização de dispositivos, dado que o objetivo dessa técnica é aumentar a eficiência na utilização dos recursos.

Além disso, a virtualização possibilita o compartilhamento do dispositivo entre várias máquinas virtuais, possibilitando sua utilização em projetos como o *SETI@home* (ANDERSON et al., 2002), que utilizam o poder computacional disponível nas máquinas de usuários para a execução de tarefas. Uma vez que a GPU está disponível para a utilização na maior parte do tempo, é possível sua utilização nesse tipo de tarefa.

Apesar disso, é possível observar oportunidades de otimizações, que servem como trabalhos futuros. Um dessas otimizações seria a utilização do HGSMI para a transferência de memória. Essa abordagem parece acrescentar menos *overheads* às transferências de dados para o dispositivo. Além disso, é possível a utilização da solução proposta com outros *frameworks* de GPGPU, possibilitando que um maior número de aplicações possa ser executado em máquinas virtuais.

REFERÊNCIAS

ANDERSON, D. et al. SETI@ home: an experiment in public-resource computing. **Communications of the ACM**, [S.l.], v.45, n.11, p.56–61, 2002.

DUNCAN, R. A Survey of Parallel Computer Architectures. **Computer**, [S.l.], v.23, p.5–16, 1990.

GOLDCHLEGER, A. et al. InteGrade: object-oriented grid middleware leveraging the idle computing power of desktop machines. **Concurrency and Computation: Practice & Experience**, [S.l.], v.16, n.5, p.449–459, 2004.

HARRIS, M. et al. **Physically-based visual simulation on graphics hardware**. 2002. 109–118p.

LICENSE, G. Version 2. **URL: http://www. gnu. org/licenses/gpl. txt**, [S.l.], 2006.

MUNSHI, A. The OpenCL Specification. **Khronos OpenCL Working Group**, [S.l.], p.l1–15, 2009.

NVIDIA. Nvidia CUDA Library Documentation. **http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/index.html (Junho de 2010)**, [S.l.], 2010.

NVIDIA, C. Compute Unified Device Architecture Programming Guide. **NVIDIA: Santa Clara, CA**, [S.l.], 2007.

PHARR, M.; FERNANDO, R. Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation. **NVIDIA: Santa Clara, CA**, [S.l.], 2005.

PURCELL, T. et al. **Ray tracing on programmable graphics hardware**. 2005. 268p.

RODRIGUES, A.; GOUVÊA, T.; PEREIRA, F. Cluster e Grid Computing. , [S.l.], 2009.