MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE MESTRADO EM SISTEMAS E COMPUTAÇÃO

LAION LUIZ FACHINI MANFROI

AVALIAÇÃO DE ARQUITETURAS *MANYCORE* E DO USO DA VIRTUALIZAÇÃO DE GPUS EM AMBIENTES DE HPDC

Rio de Janeiro
2014

INSTITUTO MILITAR DE ENGENHARIA

LAION LUIZ FACHINI MANFROI

AVALIAÇÃO DE ARQUITETURAS *MANYCORE* E DO USO DA VIRTUALIZAÇÃO DE GPUS EM AMBIENTES DE HPDC

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Sistemas e Computação.

Orientadores:
Prof. Bruno Richard Schulze - D.Sc.
Profª Raquel Coelho Gomes Pinto - D.Sc.

Rio de Janeiro
2014

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80-Praia Vermelha
Rio de Janeiro-RJ CEP 22290-270

003.54 Manfroi, Laion Luiz Fachini.

M276a Avaliação de arquiteturas *manycore* e do uso da virtualização de GPUs em ambientes de HPDC/ Laion Luiz Fachini Manfroi, orientado por Raquel Coelho Gomes Pinto e Bruno Richard Schulze. – Rio de Janeiro: Instituto Militar de Engenharia, 2014.

67 p.: il.

Dissertação (mestrado) – Instituto Militar de Engenharia – Rio de Janeiro, 2014.

1. Engenharia de sistemas e computação – teses, dissertações. 2. Sistemas de Computação. 3. Arquitetura de computadores. I. Pinto, Raquel Coelho Gomes. II. Schulze, Bruno Richard. III. Título. IV. Instituto Militar de Engenharia.

CDD 003.54

INSTITUTO MILITAR DE ENGENHARIA

LAION LUIZ FACHINI MANFROI

AVALIAÇÃO DE ARQUITETURAS *MANYCORE* E DO USO DA VIRTUALIZAÇÃO DE GPUS EM AMBIENTES DE HPDC

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Sistemas e Computação.

Orientadores: Prof. Bruno Richard Schulze - D.Sc.
Profª Raquel Coelho Gomes Pinto - D.Sc.

Aprovada em 30 de Janeiro de 2014 pela seguinte Banca Examinadora:

Profª Raquel Coelho Gomes Pinto - D.Sc. do IME - Presidente

Prof. Bruno Richard Schulze - D.Sc. do LNCC

Prof. Antonio Roberto Mury - D.Sc. do LNCC

Prof. Anderson Fernandes Pereira dos Santos - D.Sc. do IME

Prof. Lauro Luis Armondi Whately - D.Sc. da UFRJ

Rio de Janeiro
2014

Dedico esta à minha família, por ter me dado todo o apoio possível e também à minha namorada Lúcia, pelo seu amor, carinho e compreensão.

AGRADECIMENTOS

Primeiramente agradeço à Deus pela minha vida, por guiar todos os meus passos e por proteger as pessoas maravilhosas com quem convivo.

À minha família, em especial a minha namorada Lúcia, por todo o seu apoio, compreensão e paciência nos momentos mais difíceis desta caminhada.

Agradeço aos professores do Instituto Militar de Engenharia pelos ensinamentos durante todo o período de disciplinas e elaboração desta dissertação. Em especial, à professora Raquel Coelho que, juntamente ao professor Bruno R. Schulze, sempre demonstraram sua confiança neste trabalho, desde a aquisição de recursos computacionais até a disponibilidade para reuniões, orientando e oferecendo todo o suporte para que este projeto pudesse ser realizado e concluído.

Gostaria de agradecer a todos os envolvidos nos projetos ComCiDis e INCT-MACC, especialmente ao Professor Antonio Roberto Mury por sua amizade, conselhos, ensinamentos e experiências transmitidos na elaboração deste trabalho.

Por fim, a todos os professores e funcionários da Seção de Engenharia de Computação (SE/8) do Instituto Militar de Engenharia.

Laion Luiz Fachini Manfroi

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS

ABREVIATURAS

AMD	-	*Advanced Micro Devices*
CPU	-	*Central Processing Unit*
DMA	-	*Direct Memory Access*
DMAR	-	*DMA Remapping*
E/S	-	*Entrada e Saída*
GF	-	*Gigaflop*
GPGPU	-	*General-Purpose Graphics Processing Units*
GPU	-	*Graphics Processing Unit*
HPDC	-	*Computação Científica Distribuída de Alto Desempenho*
HW	-	*Hardware*
IaaS	-	*Infrastructures as a Service*
IOMMU	-	*Input/Output memory management unit*
MIC	-	*Many Integrated Core Processor*
MMU	-	*Memory Management Unit*
MV	-	*Máquina Virtual*
PF	-	*Petaflop*
SO	-	*Sistema Operacional*
SM	-	*Streaming Multiprocessor*
SP	-	*Streaming Processor*
SW	-	*Software*
VDI	-	*Virtual Desktop Infrastructure*
VMM	-	*Virtual Machine Monitor*

RESUMO

Atualmente a virtualização encontra-se presente tanto nas diversas estratégias para a consolidação de recursos em *Data Centers*, como no suporte às pesquisas em Nuvens Computacionais. Ao mesmo tempo em que novos modelos de infraestruturas de HPC combinam arquiteturas de processamento *multi-core* e *manycore* (aceleradores), ainda há uma grande expectativa de como a camada de virtualização afeta o acesso a estes dispositivos e seu desempenho. Este trabalho busca estabelecer uma avaliação de desempenho dos diferentes hipervisores, quando associados ao uso destas arquiteturas *multi-core* e *manycore*. Além disto, é estabelecido um comparativo entre as diferentes arquiteturas disponíveis no mercado atual de HPC.

ABSTRACT

Nowadays virtualization is present in various strategies for resource consolidation in data centers, as in supporting research in Cloud Computing environments. At the same time, new models of HPC infrastructures combine multi-core processing and manycore architectures (accelerators), bringing great expectations how the virtualization layer affects the access to these devices and their performance. This work aims to establish a performance evaluation of different hypervisors when combined with these multi-core and manycore architectures. Moreover, it is established a comparison between different architectures available in current HPC market.

1 INTRODUÇÃO

A Computação Científica Distribuída de Alto Desempenho (HPDC) é um segmento da ciência da computação que é capaz de combinar os avanços na pesquisa de diferentes áreas (redes de alta velocidade, *software*, computação distribuída e processamento paralelo), com o objetivo principal de oferecer um ambiente de alto desempenho, capaz de prover computação em larga escala com custo efetivo. A HPDC é utilizada principalmete na resolução de problemas complexos, disponibilizando os recursos de computação necessários para tomada de decisões, inovação de produtos tecnológicos e aceleração da pesquisa e desenvolvimento. Sua principal estratégia é dividir e distribuir a carga de processamento entre diversos computadores.

A exigência de um maior desempenho das aplicações faz com que a HPDC esteja sempre em constante evolução. No passado, a base da computação distribuída de alto desempenho eram processadores de uso genérico, ou seja, os processadores que executavam todos tipos de instruções e não eram dedicados. Atualmente, os processadores gráficos (GPUs - *Graphics Processing Units*) são os responsáveis pelo alto desempenho de grande parte dos supercomputadores mais poderosos (TOP500, 2013).

As GPUs modernas são capazes de oferecer um poder de processamento de diversas ordens de magnitude maiores que as CPUs de propósito geral. Porém, chegar nessa escala de PFs (Petaflops - 1 quadrilhão de operações de ponto flutuante por segundo) só foi possível com a combinação de CPUs com múltiplos núcleos (*multi-core*) e GPUs com muitos núcleos (*manycore*). No mesmo momento de crescimento do uso de GPUs para HPDC, a Intel® ressucita o uso dos coprocessadores com o lançamento do Intel® MIC (*Many Integrated Core Processor*), uma arquitetura de multiprocessadores direcionados à aplicações com alto processamento paralelo, combinando muitos núcleos de CPU Intel® em um único chip, com a portabilidade para códigos desenvolvidos primeiramente para a arquitetura Intel® Xeon™, formando uma arquitetura mais híbrida para desenvolvedores.

Contudo, com a disponibilidade destas diversas infraestruturas de computação heterogêneas, é necessário que seja feito um uso eficiente dos recursos disponibilizados, otimizando o tempo e a manutenção deste ambiente de processamento.

Uma maneira prática de gerenciar ambientes de alto desempenho destinados a multi-usuários envolve o uso da virtualização e o conceito de computação em nuvem. Esta infraestrutura provê benefícios, tais como abstração de recursos para o usuário, infraestrutura elástica, orientada a serviço, gerenciamento de recursos facilitado e dinamismo na disponibilização de ambiente de desenvolvimento. A interligação entre o modelo de computação distribuída de alto desempenho e a computação em nuvem baseada na virtualização é confirmada por diferentes serviços como Cluster de GPUs da Amazon EC2 (AMAZON, 2013), Nimbix (NIMBIX, 2013) e o Hoopoe (HOOPOE, 2013). Todavia, o uso de GPUs em clusters e a pesquisa do seu uso em ambientes de nuvem ainda está em seu estágio inicial, pois sempre existiu uma grande barreira no que diz respeito ao acesso das Máquinas Virtuais (MVs) às GPUs.

Este cenário começa a mudar a partir do desenvolvimento da tecnologia IOMMU (*Input/Output memory management unit*), uma função no *chipset* que traduz endereços usados em transações DMA (*Direct Memory Access*) e protege a memória de acessos ilegais direcionados de dispositivos de Entrada e Saída (E/S). Com este tipo de tecnologia, é possível dedicar um dispositivo de E/S (placas de vídeo, placas de rede, teclado e mouse) exclusivamente a uma MV, com acesso direto e exclusivo ao dispositivo. Sendo assim, é possível fazer o melhor uso dos recursos físicos de um servidor hospedeiro, disponibilizando diversas MVs com diferentes finalidades e acessando diretamente suas placas físicas, acarretando em uma menor perda de desempenho.

1.1 OBJETIVOS E CONTRIBUIÇÕES

A principal contribuição deste trabalho é comparar as arquiteturas paralelas (GPUs e Co-processadores) com uma arquitetura de CPUs *multi-core* tradicional. Isto será feito com o intuito de estabelecer o quantitativo de ganho que pode ser alcançado quando executando uma aplicação científica em determinada arquitetura. Além disso, será feira uma análise do atual estado da arte da virtualização de GPUs, através da implantação de um sistema de HPDC baseado nesta infraestrutura. Com isso, profissionais que necessitem implantar uma infraestrutura semelhante serão capazes de verificar a solução de virtualização mais adequada.

O objetivo deste trabalho é a pesquisa e o desenvolvimento de testes na implantação de um ambiente para execução de GPUs virtualizadas, fazendo uso do atual estado da arte da tecnologia de IOMMU. A princípio, este ambiente será baseado em duas soluções de

virtualização que oferecem a capacidade de acesso direto ao dispositivo de E/S (IOMMU).

Além disso, por meio de testes nestes ambientes, será realizado um comparativo do uso de GPUs em máquinas reais e em máquinas virtuais, através dos *Dwarfs*, um novo conceito utilizado na análise de desempenho. A partir desta análise, outro objetivo deste trabalho é investigar como cada solução de virtualização trata o acesso direto a determinado dispositivo de E/S através do IOMMU.

Com esta análise de desempenho, será obtida a base para definir que tipo de arquitetura (baseada em GPU ou *"big-cores"* x86) e qual solução de virtualização será mais apropriada para cada tipo de aplicação definida pelos *Dwarfs*.

A implantação deste ambiente e esta análise serão executados utilizando a infraestrutura do grupo de Computação Científica Distribuída (ComCiDis (COMCIDIS, 2013)), situado no Laboratório Nacional de Computação Científica (LNCC).

1.2 ORGANIZAÇÃO DO TRABALHO

A organização do presente trabalho encontra-se da seguinte forma: no capítulo 2 são apresentadas as bases teóricas para este trabalho, bem como os paradigmas baseados em ambientes de HPDC; no capítulo 3 é detalhada a taxonomia dos *Dwarfs* (método de classificação de aplicações científicas utilizado), assim como a revisão da bibliografia é apresentada no capítulo 4; no capítulo 5 é detalhada a análise comparativa proposta, com as tecnologias que a compõem, além da escolha dos testes a serem feitos; no capítulo 6 são expostos os resultados obtidos e suas análises nos ambientes reais e virtuais, bem como os comentários sobre os mesmos; por fim, no capítulo 7 são apresentadas as considerações finais acerca deste trabalho.

2 CONCEITOS BÁSICOS

Neste capítulo as bases teóricas e aspectos relacionados aos objetivos deste trabalho são discutidos, além de oferecer as respectivas revisões bibliográficas.

2.1 COMPUTAÇÃO CIENTÍFICA DISTRIBUÍDA DE ALTO DESEMPENHO

A frequente necessidade de maior capacidade computacional em sistemas de computação intensiva faz com que os processadores e outros componentes tenham que trabalhar cada vez com mais rapidez. Porém, limites físicos ainda existem na computação, tais como: termodinâmica, limites de armazenamento, limites de comunicação e de frequência. Uma forma de contornar essa restrição é a utilização de técnicas que possibilitam o processamento distribuído. A grande meta da agregação de recursos computacionais é prover respostas para as limitações encontradas nas arquiteturas centralizadas utilizando técnicas que possibilitem o processamento distribuído.

A computação distribuída em larga escala é, usualmente, denominada de Computação Científica Distribuída de Alto Desempenho (HPDC) e pode ser entendida como uma área da ciência da computação que tem como objetivo a melhoria do desempenho de aplicações distribuídas e paralelas, utilizando-se de complexas infraestruturas computacionais (DANTAS, 2005). Já a Computação Científica pode ser entendida como a interseção da modelagem de processos científicos, com a utilização de computadores para produzir resultados quantitativos a partir desses modelos (EIJKHOUT, 2013).

A constante evolução da HPDC faz com que cada vez mais aplicações necessitem de sua integração. As análises militares e de previsões meteorológicas são um exemplo do seu uso, além de poder ser aplicada na área de finanças, medicina, farmacologia, biologia e aplicações médicas. Além disso, através da HPDC é possível alcançar o processamento de grandes volumes de dados experimentais criados por uma nova geração de instrumentos e aplicações científicas, que exigem grande largura de banda, redes de baixa latência e recursos computacionais de alto desempenho (HEY, 2009). A evolução da HPDC também permitiu que novas estratégias e tecnologias fossem adicionadas à sua perspectiva de processamento, auxiliando na maneira com que determinados dados são tratados e processados, otimizando ainda mais o seu uso. No passado, a base da computação de alto

desempenho eram processadores de uso genérico, ou seja, os processadores que executavam todos tipos de instruções, aqueles que não são dedicados a um único tipo de aplicação. Em 2008, o RoadRunner (IBM) superou a barreira de 1 PF (1 quadrilhão de operações de ponto flutuante por segundo) desenvolvendo o processador IBM PowerXCell 8i, baseando-se na GPU Cell (Playstation 3). Atualmente, os processadores gráficos são os responsáveis pelo alto desempenho da maior parte dos supercomputadores (TOP500, 2013), devido à sua capacidade de processar paralelamente os dados. A seguir são apresentados alguns aspectos importantes e o atual estado da arte da computação paralela, juntamente às GPUs e aceleradores *manycore* e arquiteturas de CPUs e *multi-core*.

2.2 ARQUITETURAS DE PROCESSAMENTO PARALELO

Uma arquitetura paralela fornece uma estrutura explícita e de alto nível para o desenvolvimento de soluções utilizando o processamento paralelo, através da existência de múltiplos processadores que cooperam para resolver problemas, através de execução concorrente (DUNCAN, 1990).

O surgimento da computação paralela deu-se pela necessidade de aumentar a potência computacional, com o intuito do alcance do alto desempenho para aplicações específicas. Adicionalmente, o objetivo se baseava em solucionar grandes problemas com um tempo de processamento menor do que aquele realizado por computadores seqüenciais (arquitetura de *von Neumann*), tendo vários processadores em uma única máquina, cooperando e havendo comunicação entre si. Como um aumento expressivo de desempenho nas arquiteturas de *von Neumann* sempre foi uma tarefa árdua devido às limitações tecnológicas, a computação paralela tem sido tratada como uma alternativa atrativa, particularmente quando os problemas a serem solucionados são essencialmente paralelos.

Várias mudanças ocorreram na área de computação nas últimas décadas, levando à alta conectividade dos recursos computacionais, o que permitiu a solução de vários problemas de modo mais eficiente, a um custo relativamente mais baixo.

Com o avanço da computação paralela, foram propostas várias maneiras de conexão entre os recursos computacionais, criando diferentes arquiteturas paralelas. Cada arquitetura apresenta determinadas características, visando melhor desempenho sob um dado enfoque. Para acompanhar o desenvolvimento das arquiteturas paralelas e agrupar os equipamentos com características comuns, foram propostas algumas taxonomias, dentre elas a de Flynn (FLYNN, 1972) e a de Duncan (DUNCAN, 1990). Entre estas, a

taxonomia de Flynn é a mais usada, sendo baseada em dois conceitos: fluxo de instruções (contador de programa) e fluxo de dados (conjunto de operandos). Como estes fluxos são independentes, existem quatro combinações entre eles:

- SISD (*Single Instruction Single Data*) - Clássico computador sequencial de Von Neumann. Um fluxo de instrução, um fluxo de dados e faz uma coisa por vez;
- SIMD (*Single Instruction Multiple Data*) - Execução síncrona de instrução para todos os dados. Um fluxo de instrução e múltiplos fluxos de dados. Usado em arquiteturas vetoriais, onde a mesma instrução é executada sobre múltiplos operandos, em GPUs e em coprocessadores com a arquitetura Intel® MIC (MACKAY, 2013);
- MISD (*Multiple Instruction Single Data*) - Múltiplas instruções operando no mesmo dado. Não é claro se tais máquinas existem, porém alguns consideram como MISD as máquinas com pipeline;
- MIMD (*Multiple Instruction Multiple Data*) - Múltiplas CPUs independentes operando em múltiplos dados. A maioria dos processadores paralelos cai nesta categoria;

Desde os anos 80, vários trabalhos foram desenvolvidos com o objetivo de explorar o potencial dos sistemas computacionais distribuídos aliado aos conceitos da computação paralela. Sendo assim, a convergência das áreas de computação paralela e de sistemas distribuídos trouxe uma nova expectativa de vantagens, principalmente no que se refere à implementação da computação paralela, proporcionando redução de custos e a utilização mais adequada de recursos computacionais. Com isso, foi possível a união do custo relativamente baixo oferecido pelos sistemas computacionais distribuídos, ao alto desempenho fornecido pelo processamento paralelo, originando o que passou a ser conhecido como "Computação Paralela Distribuída", que compõe o ambiente de HPDC.

O alto custo dos equipamentos com arquiteturas paralelas e da implantação desses sistemas sempre representou um obstáculo à sua ampla disseminação. Por outro lado, o desempenho dos computadores pessoais e das estações de trabalho tem apresentado um aumento significativo ao longo das últimas décadas. Ao mesmo tempo, o custo relativamente baixo dessas máquinas favorece a sua ampla utilização, levando à interconexão destes equipamentos, permitindo o estabelecimento de sistemas computacionais

distribuídos, constituindo uma área amplamente difundida e pesquisada nas últimas três décadas (COULOURIS, 1994).

A partir desta grande pesquisa na área de HPDC, foi possível chegar na escala dos Petaflops através da combinação de CPUs com múltiplos núcleos (*multi-core*) e GPUs com muitos núcleos (*manycore*), atingindo o auge do uso da computação nas diversas áreas da ciência (KIRK, 2011). Devido a isto, os processadores *manycore* evoluíram e vêm se tornando, cada vez mais, uma parte importante do ambiente de computação de alto desempenho. A seguir são apresentadas duas arquiteturas de dispositivos que serão utilizados neste trabalho, as GPUs e os coprocessadores baseados na arquitetura Intel® MIC.

2.2.1 GPU - *GRAPHICS PROCESSING UNITS*

Por mais de duas décadas, o aumento de desempenho e a redução nos custos das aplicações foram impulsionadas por microprocessadores baseados em uma única unidade central de processamento. As CPUs também foram responsáveis pelo rompimento da barreira dos GFs (bilhões de operações de ponto flutuante por segundo) em *desktops* e centenas de GFs em servidores em cluster. Porém, este impulso caiu em 2003, onde as questões de consumo de energia e de dissipação do calor limitaram o aumento da frequência do *clock*, além do nível de tarefas que podiam ser realizadas em cada período em uma única CPU. Neste período, os fabricantes de microprocessadores passaram a se basear em modelos em que várias unidades de processamento (núcleos) são usadas em cada chip para aumentar o poder de processamento, exercendo um grande impacto sobre a comunidade de desenvolvimento de SW.

Tradicionalmente, a maioria das aplicações é escrita sequencialmente. Os usuários se acostumaram a esperar que estas aplicações executem mais rapidamente a cada nova geração de microprocessadores, sendo uma expectativa inválida daqui pra frente. Ao contrário, as aplicações que continuarão a ter melhoria de desempenho serão as aplicações baseadas em arquiteturas paralelas, com várias *threads* de execução que cooperam entre si. Desde então, a indústria tem estabelecido duas trajetórias principais para o projeto de microprocessador: *multi-core* (múltiplos núcleos) e *manycore* (muitos núcleos).

A trajetória *multi-core* busca manter a velocidade de execução dos programas sequenciais, enquanto se move por múltiplos núcleos sempre projetados para maximização da velocidade de execução dos programas sequenciais.

A trajetória *manycore* é focada na execução de várias aplicações paralelas, contendo um grande número de núcleos muito menores. As GPUs são arquiteturas *manycore*, e têm liderado a corrida do desempenho em ponto flutuante desde 2003, como ilustrado na figura 2.1. Pode-se dizer que entre 2008 e 2009 a razão entre GPUs *manycore* e CPUs *multi-core* era de 10 para 1 (KIRK, 2011). Por essa razão, muitos desenvolvedores de aplicações, além da comunidade científica em geral, vêm adotando o uso de GPUs e migrando os trechos computacionalmente intensos de seus SWs. A principal diferença de desempenho entre GPUs e CPUs deve-se às filosofias com que cada unidade é desenvolvida.

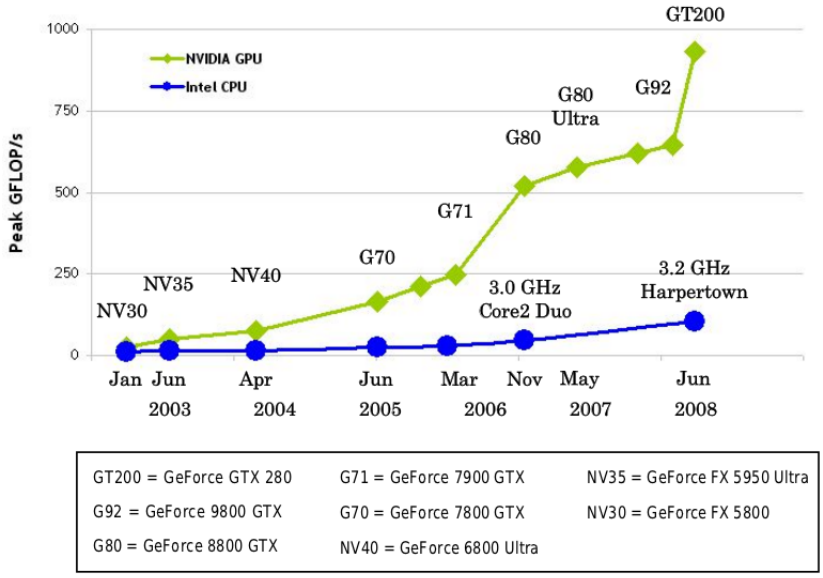

FIG. 2.1: Crescente diferencial de performance entre GPUs e CPUs (KIRK, 2011).

As CPUs são desenvolvidas com múltiplos núcleos e com a filosofia de uso geral, onde todos os tipos de aplicações devem ser executadas nela, além de oferecer o controle das instruções do Sistema Operacional (SO) nativo da máquina e operar as diversas solicitações dos periféricos de E/S, sendo otimizada para o uso sequencial.

O desenvolvimento das GPUs é impulsionada e modelada pela crescente indústria de jogos, onde o importante é realizar um número alto de cálculos de ponto flutuante por quadro, motivando os fabricantes de GPUs a maximizar a área do chip e o consumo de energia dedicados aos cálculos de ponto flutuante. Pode-se verificar a maximização da área do chip na figura 2.2. Logo, as GPUs são projetadas como mecanismo de cálculo numérico, e elas não funcionam bem em algumas tarefas em que as CPUs são naturalmente projetadas para funcionar bem. Com isso, a maioria das aplicações usará

tanto CPUs (partes sequenciais) quanto GPUs (partes numericamente intensivas).

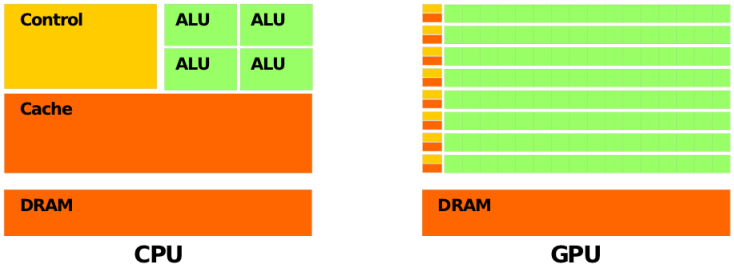

FIG. 2.2: Filosofia de projetos diferentes entre GPUs e CPUs (KIRK, 2011).

A figura 2.3 demonstra uma arquitetura genérica de GPUs, que é organizada como uma matriz de multiprocessadores de *streaming* (SMs) altamente encadeados. Pode-se verificar que dois SMs formam um bloco, no entanto, o número de SMs em um bloco varia entre as gerações de GPUs. Cada SM possui um determinado número de processadores de streaming (SPs), que compatilham a lógica de controle e a cache de instruções. As GPUs também possuem a memória global (*global memory*) que é usada exclusivamente para os gráficos e cálculos da GPU. Para aplicações gráficas, a memória global guarda imagens e informações de textura, mas no cálculo ela funciona como uma memória externa de alta largura de banda. Cada SP é maciçamente encadeado (*threaded*) e pode executar milhares de *threads* por aplicação. Geralmente, uma aplicação paralelizada executa de 5000 a 12000 *threads* simultaneamente na GPU. O número de *threads* admitidas por núcleos varia entre gerações de GPU, logo o nível de paralelismo admitido pelas GPUs está aumentando rapidamente, o que a faz cada vez mais presente no estudo de aplicações paralelas.

Pode-se dizer que as GPUs cada vez mais irão trabalhar como coprocessadores, auxiliando cada vez mais a CPU em aplicações que deverão ser paralelizadas. Esta visão da GPU como coprocessador fez com que a fabricante Intel® ressurgisse com o conceito dos coprocessadores em sua mais nova arquitetura, descrita a seguir.

2.2.2 COPROCESSADORES E A ARQUITETURA INTEL MIC

Os coprocessadores foram desenvolvidos primeiramente para computadores de grande porte com a finalidade de incluir uma CPU para complementar as funções da CPU principal, auxiliando nas operações de ponto flutuante e outras que necessitam de processamento aritmético intensivo. Devido a isto, primeiramente ele foi denominado coprocessador aritmético (MORIMOTO, 2013). Os coprocessadores podem acelerar o desempenho geral

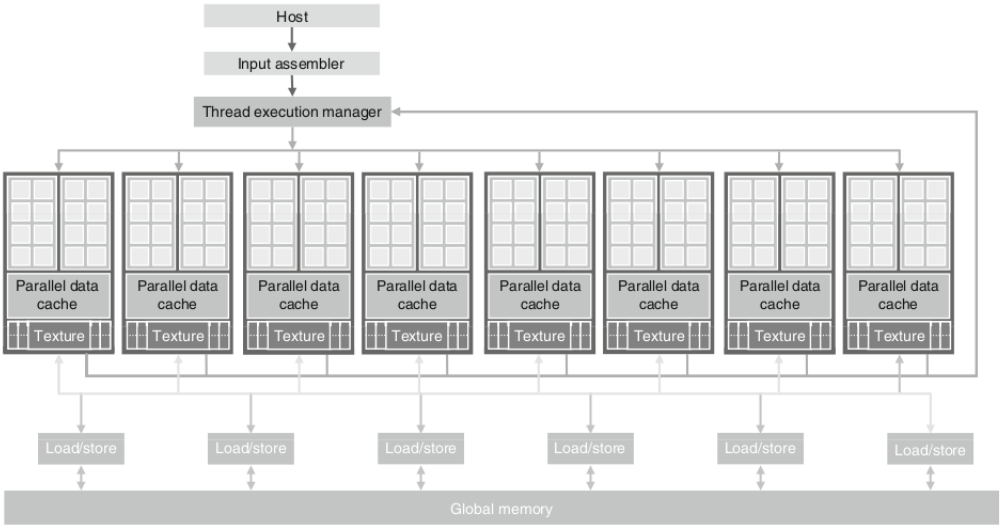

FIG. 2.3: Arquitetura de GPUs (KIRK, 2011).

de um sistema, permitindo sua aquisição somente pelos usuários que necessitem do seu processamento intensivo. Na época de seu lançamento, sua estratégia de uso não propiciou sua ampla difusão como um processador auxiliar, pois poucos usuários equipavam seus computadores com estes chips, acarretando na baixa venda e, consequentemente, no aumento de preço. Estes equipamentos chegaram a ponto de custar mais caro que a CPU principal. Pouco tempo depois, graças às aplicações que necessitavam de seu uso e à miniaturização, os coprocessadores foram incluídos dentro do chip da CPU principal. Isto resolveu o problema do custo de produção dos coprocessadores, barateando o conjunto.

Ao mesmo tempo que a computação em GPUs vem se tornando um fator importante na comunidade científica, a Intel® anunciou o lançamento do Intel® MIC (*Many Integrated Core Processor*). Esta nova arquitetura se baseia em multiprocessadores direcionados a aplicações com alto processamento paralelo, combinando muitos núcleos de CPU Intel® em um único chip, com a vantagem que os mesmos SWs desenvolvidos para arquiteturas Intel® MIC podem ser compilados e executados em arquiteturas padrões Intel® Xeon™. Isto forma uma arquitetura mais híbrida para o desenvolvimento de aplicações e testes de desempenho (MIC, 2013). Em 18 Junho de 2012, a Intel anunciou que o nome da família de todos produtos baseados na arquitetura MIC será Xeon Phi™ (PHI, 2012), que busca trazer à ativa o conceito dos coprocessadores, juntamente à uma resposta ao tão crescente mercado de GPGPU (*General-Purpose Graphics Processing Units*).

Como dito anteriormente, as GPUs modernas são capazes de produzir um poder de

processamento de diversas ordens de magnitude maiores que as CPUs de propósito geral. Segundo a própria empresa fabricante de processadores, a expectativa é posicionar a sua nova família de coprocessadores Xeon Phi™ no mercado dos *manycore* e atingir à marca dos Hexaflops (1 Quinqualhão de operações de ponto flutuante por segundo) até 2018 (INTEL, 2011), sendo através da combinação dos processadores Xeon™ e dos coprocessadores Xeon Phi™ que se espera chegar a esta marca.

Os coprocessadores Xeon Phi™ são basicamente compostos por núcleos de processamento, memórias cache, controladores de memória (*memory controllers*), *PCIe client logic*, e um "anel" (*ring*) de conexão bidirecional de alta largura de banda, conforme ilustrado na figura 2.4. Cada núcleo possui uma cache L2 privada que é mantida completamente coerente com o diretório de TAGs *TD - Tag Directory*. Os endereços de memória são distribuídos uniformemente pelos TDs, logo, se um núcleo recebe um *cache miss* e necessita de um bloco de memória presente em uma cache de outro núcleo, ele é repassado, porém se o bloco não é encontrado em nenhuma cache, um endereço de memória é repassado do TD ao controlador de memória. O controlador de memória é a interface direta com a memória principal do coprocessador, e o *PCIe client logic* é a interface direta com o barramento PCIe. Todos estes componentes são conectados pela sua estrutura em anel (*ring*).

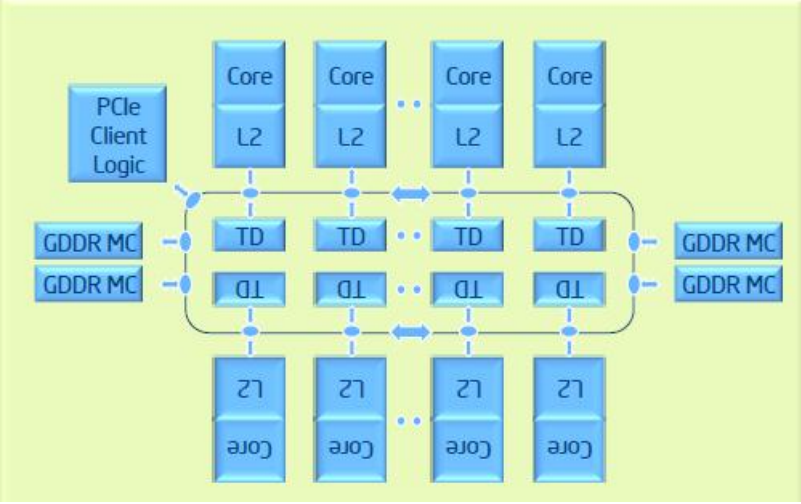

FIG. 2.4: Microarquitetura Xeon Phi™ (MIC, 2013).

Esta reinvenção dos coprocessadores separados pode ser uma estratégia para minimizar os problemas de dissipação de calor, aumentando a velocidade dos processadores sem colocar em risco os chips. Ao mesmo tempo, é uma maneira de disputar o campo de

processamento paralelo junto às GPUs.

Com a disponibilidade destas diversas infraestruturas de computação heterogêneas, é necessário que seja feito um uso eficiente, otimizando o tempo e os recursos disponíveis para este processamento. Na próxima subseção será discutido o uso da virtualização na computação de alto desempenho e como o atual estado da arte pode ajudar no desenvolvimento de testes e análise de desempenho nestas diversas infraestruturas disponíveis.

2.3 VIRTUALIZAÇÃO

A virtualização foi introduzida na década de 60 (CREASY., 1981), mas ressurgiu na última década tanto na indústria como na comunidade de pesquisa. A virtualização é baseada na utilização de máquinas virtuais (MVs), um ambiente que provê interfaces de HW virtualizadas adicionando uma camada de comunicação denominada *Virtual Machine Monitor* (VMM) ou comumente chamada de hipervisor (*hypervisor*). A virtualização oferece um ambiente onde diferentes MVs (convidados) executam no mesmo servidor físico (hospedeiro), acessando o mesmo HW, proporcionando o uso mais eficiente dos recursos disponíveis no servidor. A portabilidade também é um dos pontos fortes da virtualização, isto se dá pela capacidade de ambientes facilmente serem migrados e manuseados entre diversos servidores, obedecendo a demandas de recursos mais poderosos, atualizações de HW e acordos de níveis de serviço. Embora a virtualização fosse originalmente focada para compartilhamento de recursos, atualmente uma ampla variedade de benefícios podem ser alcançados com o seu uso, tais como:

- Facilidade no gerenciamento de recursos - Isto se dá pela capacidade de abstração dos recursos de maneira "virtual", economizando tempo de implantação de novo HW ou SW, instalação de SOs e configuração;

- Isolamento e Segurança - O isolamento é a capacidade de uma MV operar em um ambiente independente do SO nativo. Sendo assim, o que é feito dentro da MV, ficará restrito a este ambiente sem interferir nas operações no SO nativo do seu hospedeiro;

- Salvamento de estados - Capacidade da interrupção de operação de uma MV (pausa) e salvamento de seu estado para retomada posterior (resumo);

- Migração em tempo-real (*live-migration*) - Sendo semelhante ao salvamento de estados, porém a MV é migrada de um hospedeiro a outro, em tempo-real. Esta capacidade facilita o gerenciamento e a manutenção. Um exemplo ocorre no caso de algum hospedeiro demonstrar uma queda significante de desempenho, as MVs que nele executam podem ser migradas para outros servidores sem parar sua execução;

- Alta disponibilidade - Capacidade de manter uma MV executando 24/7 (24 horas, por 7 dias da semana), disponível em 99,9% deste tempo;

Apesar de todas estas características provenientes do seu uso, diversos problemas relacionados à virtualização de CPU foram pesquisados e solucionados e atualmente existem diversas técnicas (em níveis de SW e HW) para virtualizar a CPU com baixa perda de desempenho (ADAMS, 2006). Por outro lado, a virtualização de dispositivos de E/S ainda é uma área que oferece vários problemas a serem solucionados e uma grande variedade de estratégias a serem usadas. O uso das GPUs em conjunto com a virtualização ainda apresenta desafiadoras pesquisas, devido às suas limitadas documentações.

Como dito anteriormente, as GPUs modernas são desenvolvidas para maximizar o uso do tamanho físico do chip para cálculos numéricos. Isto acarreta na presença de um número maior de transistores, pois entregam mais poder e possuem desempenho computacional de ordens de magnitude maiores que as CPUs. O aumento das aplicações que utilizam a aceleração oferecida pela GPU faz com que seja de extrema importância a pesquisa do *hardware* gráfico em ambientes virtualizados. Além disto, infraestruturas de desktop virtuais (*Virtual Desktop Infrastructure* - VDI) permitem que várias empresas simplifiquem o gerenciamento de estações de trabalho entregando MVs aos seus usuários. A virtualização de GPUs é muito importante para usuários que usam suas estações de trabalho com uma MV.

GPUs sempre foram um campo delicado de tratar na virtualização. Isto se deve graças à maneira que a multiplexação do HW real é feita. Cada MV possui um dispositivo virtual e combina suas respectivas operações no hipervisor de maneira a usar o HW nativo, preservando a ilusão que cada sistema convidado possui seu próprio dispositivo real. Porém, as GPUs são dispositivos extremamente delicados de serem tratados. Além disto, diferentemente das CPUs, dos *chipsets*, e de populares controladores, algumas características dos projetos das GPUs são proprietárias e confidenciais, não oferecendo a documentação apropriada para o uso pela comunidade científica. Adicionado ao fato de

que as arquiteturas de GPU mudam constantemente, se comparado às arquiteturas de CPUs e outros periféricos, o estudo da virtualização de GPUs modernas tem sido uma área delicada de pesquisa. Mesmo iniciando uma implementação completa, a atualização do desenvolvimento de estratégias para cada geração de GPUs não seria viável.

Este cenário começa a mudar a partir do desenvolvimento da tecnologia IOMMU (*Input/Output memory management unit*), e da funcionalidade de *PCI passthrough*, presente em diversos hipervisores comerciais e de licença livre. Atualmente os hipervisores de código aberto que empregam esta tecnologia são o XEN e o KVM, descritos nas subseções a seguir, juntamente às tecnologias citadas.

2.3.1 KVM - *KERNEL-BASED VIRTUAL MACHINE*

O KVM (KVM, 2011) é uma camada de virtualização atualmente integrada no kernel para o HW x86 do Linux. Foi o primeiro hipervisor a fazer parte do kernel nativo do Linux (2.6.20), sendo desenvolvido primeiramente por Avi Kivity, da extinta empresa Qumranet, atualmente de propriedade da Red Hat (QUMRANET, 2011).

O KVM é implementado como um módulo do kernel, permitindo que o Linux se torne um hipervisor apenas ao carregar seu módulo. O KVM oferece virtualização total em plataformas de HW (*full virtualization*), além de suportar MVs para-virtualizados (com alterações nos SOs virtualizados).

O KVM é implementado como dois componentes principais de operação para trabalhar com estes tipos de virtualização. O primeiro componente é o módulo carregável pelo KVM, que fornece gerenciamento de HW de virtualização (extensões), expondo os recursos do hospedeiro através do sistema de arquivos (figura 2.5). O segundo componente é a emulação de plataformas, que é fornecido por uma versão modificada do emulador QEMU, que é executado como um processo do usuário, coordenando junto com o kernel as requisições MV.

Quando uma nova MV é inicializada no KVM, esta se torna um processo do SO hospedeiro e, portanto, pode ser escalonada como qualquer outro processo. Mas diferentemente dos processos do Linux, uma MV é identificada pelo hipervisor como estando no modo "convidado" (independente dos modos do kernel e do usuário).

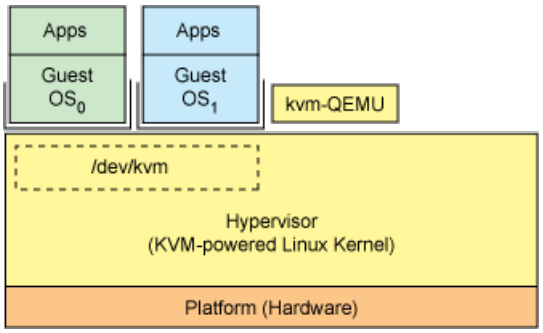

FIG. 2.5: Arquitetura KVM (KVM, 2011).

Cada MV é mapeada através de um dispositivo do KVM, e possui seu próprio espaço de endereço virtual que é mapeado para o espaço de endereço físico. O KVM usa suporte de virtualização de HW para fornecer a virtualização total, enquanto suas requisições de E/S são mapeadas através do kernel do hospedeiro para o processo do QEMU, que é controlado pelo hipervisor.

O KVM opera como um hospedeiro, mas suporta vários SOs como convidados, dando suporte para a virtualização de HW necessária, mesmo que cada SO faça uso desta de uma forma diferente.

2.3.2 XEN

O XEN (XEN, 2011) um hipervisor chamado "tipo 1", significando que ele executa diretamente no topo dos recursos físicos, criando conjuntos lógicos destes recursos do sistema, de modo que várias MVs possam compartilhá-los.

O XEN executa diretamente no HW do sistema, inserindo uma camada de virtualização entre o HW e as MVs. transformando o HW em um conjunto de recursos computacionais lógicos que podem ser alocados dinamicamente por qualquer MV. As MVs interagem com os recursos virtuais como se fossem recursos físicos (figura 2.6), isto é, a MV pode identificar os recursos reais do servidor hospedeiro como fazendo parte de seu sistema virtual.

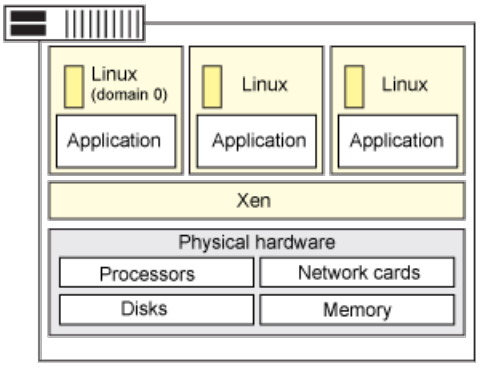

FIG. 2.6: Arquitetura XEN (XEN, 2011).

As MVs executando através do XEN são denominadas "domínios" e um domínio especial conhecido como "dom0" é responsável por controlar a inicialização do hipervisor e dos outros domínios, que são chamados de "domUs".

2.3.3 *PCI PASSTHROUGH* E IOMMU

PCI Passthrough é a capacidade de prover um isolamento de dispositivos para determinada MV, assim, o dispositivo pode ser utilizado exclusivamente por uma MV, como mostrado na figura 2.7, onde é possível verifique que, à nível de SW, o *PCI Passthrough* é a funcionalidade implementada pelo hipervisor para oferecer o acesso direto a algum dispositivo de E/S.

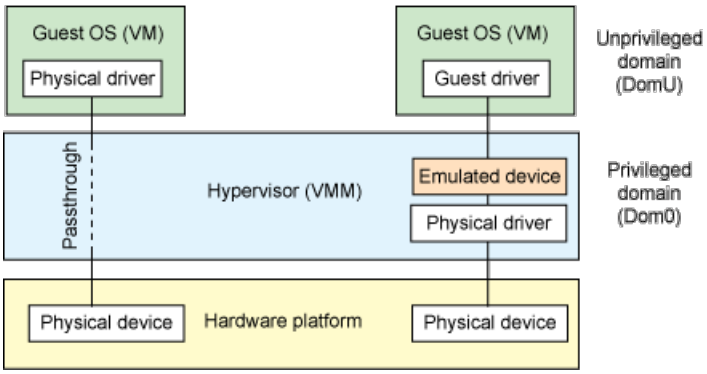

FIG. 2.7: Arquitetura PCI passthrough

A funcionalidade de prover o *PCI Passthrough* somente foi possível após o desenvolvimento de novas instruções de suporte aos hipervisores, que estão sendo integradas nos recentes HWs (processadores e placas-mãe). Além disto, a função principal desta capacidade é realizar o DMAR (DMA *remapping* - remapeamento de acessos diretos à memória

de um dispositivo), o que só foi possível através da habilitação do IOMMU.

A IOMMU é uma função no *chipset* que traduz endereços usados nas transações DMA e protege contra acessos ilegais direcionado de dispositivos de E/S. Com este tipo de tecnologia, é possível dedicar um dispositivo de E/S (placas de vídeo, placas de rede, teclado, mouse e USB *devices*) exclusivamente a uma MV, com acesso direto e exclusivo. A IOMMU vem sendo usado para diminuir a disparidade entre as capacidades de endereçamento de alguns periféricos e do processador do hospedeiro, pois estes primeiros possuem uma capacidade de endereçamento menor, fazendo com que não consigam acessar toda a memória física disponível.

A IOMMU funciona de maneira similar à MMU (*Memory Management Unit*), que é responsável por traduzir endereços lógicos (utilizados pela CPU) em endereços físicos (memória principal). A grande diferença é que a IOMMU traduz os endereços virtuais utilizados pelo dispositivo de E/S em endereços físicos, independente da tradução feita pela MMU, como ilustrado na figura 2.8.

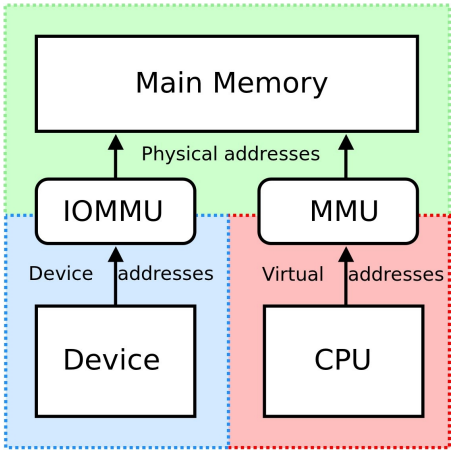

FIG. 2.8: IOMMU vs MMU (Fonte: *en.wikipedia.org/wiki/IOMMU*).

Tratando-se de um ambiente virtualizado, onde os SOs executam em MVs, a dificuldade de acesso direto à memória através do DMA (presente na maioria dos periféricos) é aumentada significativamente. Isto se deve ao momento em que o SO convidado (MV) tenta realizar uma operação DMA usando endereços físicos gerados pela MV, podendo ocorrer o corrompimento da memória, pois o HW não sabe sobre os mapeamentos executados nos endereços do hospedeiro e do convidado. Este problema é evitado pois o hipervisor ou o SO hospedeiro intervém na operação, realizando as traduções de endereços

necessárias, porém isto acarreta em uma perda significativa de desempenho nas operações de E/S. Neste caso, a IOMMU opera realizando uma melhoria de desempenho, através do DMAR, que é feito de acordo com a tabela de traduções, criada e utilizada para mapear os endereços físicos do SO hospedeiro e do convidado, conforme ilustrado na figura 2.9.

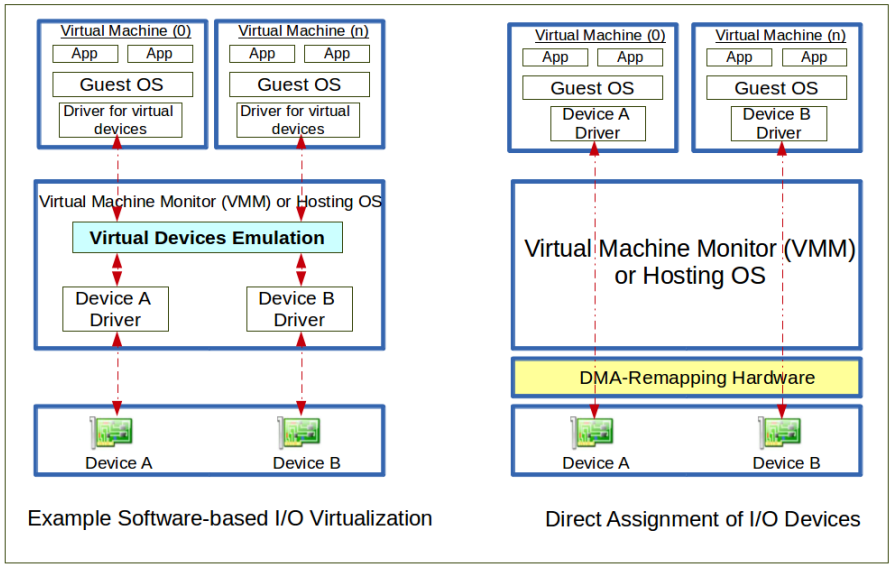

FIG. 2.9: Exemplo de virtualização utilizando IOMMU e DMA remapping.

Através da IOMMU pode-se utilizar de maneira otimizada os recursos físicos de um servidor hospedeiro, disponibilizando diversas MVs com diferentes finalidades e acessando diretamente suas placas físicas, acarretando em mínima perda de desempenho. Com isto, a tecnologia de IOMMU provê o suporte necessário para o *PCI Passthrough* através do remapeamento de DMA

Neste trabalho a IOMMU será a estratégia de baixo-nível utilizada para a virtualização das GPUs. O desenvolvimento será baseado em 2 soluções de virtualização (XEN e KVM) que implementam o acesso direto da MV à GPU.

3 APLICAÇÕES CIENTÍFICAS E DWARFS

O consumo de HW na forma de tempo de CPU, montante de memória utilizado, largura de banda de rede e espaço em disco é uma parte muito útil da informação quando disponibilizado antes da execução de uma aplicação. Podem ser utilizados por escalonadores para o maior número de aplicações sem a contenção de recursos, podendo auxiliar na estimativa do tempo de espera em sistema de execução em fila, e até mesmo prover uma estimativa do custo da execução de uma aplicação em um ambiente de nuvem (MATSUNAGA, 2010). Em adição à isto, e o mais importante do ponto de vista deste trabalho, pode identificar a melhor arquitetura e o melhor ambiente para executar uma aplicação.

Entretanto, esta informação usualmente não é disponibilizada para os usuários de sistemas computacionais que desconhecem se a arquitetura em uso (ou que será adquirida) é a melhor para maximizar a eficiência de sua aplicação. A maneira convencional de avaliar uma arquitetura é estudar uma suite de *benchmarks* baseada em programas já existentes. Contudo, o uso de um *benchmark* fornece um nível de desempenho que unicamente caracteriza a arquitetura (ou o ambiente em questão) com relação a outros sistemas, sem considerar uma perspectiva de uma aplicação. As suites de *benchmarks* convencionais podem prover uma boa estratégia de comparação entre diversas arquiteturas, mas é necessária a percepção de que por vezes seus resultados não mostram o espectro relevante por completo, ao mesmo tempo em que não possuem relação com fatores específicos da aplicação que será executada naquele ambiente. Por exemplo, o *benchmark Linpack*, ou o *benchmark* Top500 (DONGARRA, 1999) é a mais reconhecida métrica para ordenar sistemas de HPC. Entretanto, o Linpack *"está cada vez menos confiável como a única métrica de performance para uma coleção cada vez maior de aplicações científicas"* (HEROUX, 2013). Os cálculos predominantes no algoritmo presente no Linpack são multiplicações entre matrizes densas, o que favorece fortemente arquiteturas com altas taxas de processamento em ponto-flutuante. Estas características correspondem a somente uma classe de *Dwarfs*, a classe de Álgebra Linear Densa (DLA - *Dense Linear Algebra*), embora atualmente existam treze classes. Entretanto, apesar deste padrão ser comumente encontrado em aplicações no mundo real, para uma completa avaliação do atual universo de

aplicações científicas, uma cobertura adicional dos principais padrões de comunicação e computação são necessários. Os experimentos neste trabalho utilizam o *Dwarf* DLA, pois é o campo de atuação maior dentre todos, ou seja, é o que mais se adequa à determinados padrões utilizados (SHALF, 2009) em variados ramos de pesquisa (FIG 3.1). Apesar de ser o *Dwarf* que maios engloba aplicações científicas, ainda existem muitas pesquisas e avaliações a serem realizadas nesta classe.

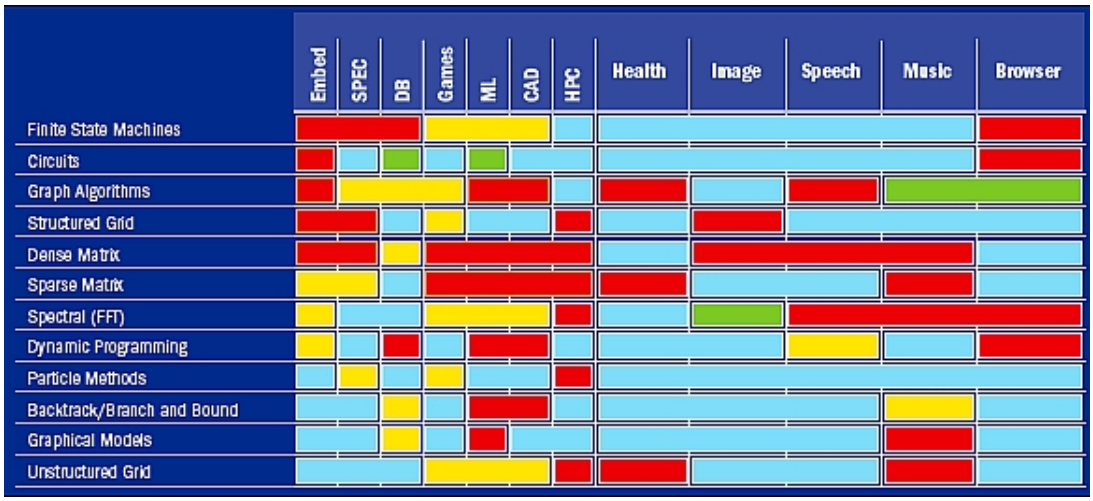

FIG. 3.1: Campo de atuação de cada *Dwarf* (SHALF, 2009).

3.1 TAXONOMIA DOS DWARFS

Com o objetivo de delinear requisitos de aplicações científicas, o trabalho de Phillip Colella (COLELLA, 2004) identificou sete métodos numéricos que, àquela altura, acreditavam ser os pontos importantes para a ciência e a engenharia, introduzindo assim, os "Sete Dwarfs" da computação científica. Estes *Dwarfs* são definidos, em um nível maior de abstração, para explicar seus comportamentos entre diferentes aplicações de HPC e cada classe de *Dwarfs* possui similaridades em computação e comunicação. De acordo com sua definição, as aplicações de uma determinada classe podem ser implementadas diferentemente com as mudanças nos métodos numéricos que ocorreram com o passar do tempo, contudo, os padrões subjacentes permaneceram os mesmos durante a geração de mudanças e permancerão os mesmos em implementações futuras.

A equipe de Computação Paralela de Berkeley estendeu estas classificações dos *Dwarfs* para treze, após examinarem importantes domínios de aplicações, com interesses na aplicação dos *Dwarfs* para um maior número de problemas computacionais (ASANOVIC,

2006). O objetivo foi definir requisitos de aplicações, com o intuito de atingir a capacidade de obter conclusões mais específicas sobre requisitos de HW. Os atuais treze *Dwarfs* estão listados na tabela 3.1, baseado no trabalho de Berkeley, com uma breve descrição (ASANOVIC, 2006), (KAISER, 2010) e (SPRINGER, 2011).

O *Dwarf* de DLA engloba os casos de uso de operadores matemáticos em escalares, vetores ou matrizes densas. A característica de ser "Denso" deste *Dwarf* se refere às estruturas de dados acessadas durante o esforço computacional realizado.

Algumas aplicações caracterizadas como como DLA são: Decomposição LU, Cholesky, Transpostas de matrizes, algoritmos de clusterização (*K-means* e *Stream Cluster*). Exemplos de áreas científicas onde seus algoritmos estão diretamente ligados com a classe DLA são: Ciências dos materiais (física molecular e nanotecnologia), setor de energia (fusão e física nuclear), ciências fundamentais (astrofísica e física nuclear) e engenharias (design para aerodinâmica).

Algumas evidências para a existência de classes equivalentes, propostas pelos *Dwarfs*, podem ser encontradas em algumas bibliotecas para cálculo numérico, tais como: LAPACK (BLACKFORD, 1996) para DLA, OSKI (VUDUC, 2006) para álgebra linear esparsa e FFTW (FRIGO, 2005) para métodos espectrais. É importante salientar que muitas aplicações podem ser caracterizadas como mais de um *Dwarf* (um exemplo é a computação gráfica, que involve álgebra linear densa e grafos transversos). Este é um dos pontos que evidenciam que a avaliação de uma arquitetura utilizando somente um *benchmark* não é a mais apropriada. *Benchmarks* são úteis, porém estes representam somente um modelo, que pode não combinar com os requisitos da aplicação que executará em uma arquitetura.

É necessário conhecer o modelo de aplicações para escolher um *benchmark* (ou um conjunto deles) mais apropriado. No próximo capítulo serão apresentados os trabalhos relacionados à classificação dos *Dwarfs* e aos modelos de avaliações utilizados neste trabalho.

TAB. 3.1: Os treze Dwarfs e suas descrições

Nome do Dwarf	**Descrição**
Álgebra Linear Densa	Os dados são matrizes ou vetores densos.
Álgebra Linear Esparsa	Matrizes e vetores incluem muitos valores zerados.
Métodos Espectrais	Geralmente involve o uso da transformada de Fourier.
N-Body	Dependência da interação entre muitos pontos considerados discretos, Cálculos Partícula-Partícula.
Structured Grids	Representados por grades regulares onde os pontos são atualizados ao mesmo tempo.
Unstructured Grids	Grades irregulres onde as localidades de dados são selecionados por características da aplicação.
Monte Carlo	Cálculos dependem de resultados estatísticos gerador por tentativas aleatórias repetitivas.
Lógica Combinacional	Funções que são implementadas com funções lógicas e estados.
Grafos Transversos	Visita de muitos nós em um grafo seguindo arestas sucessivas.
Programação Dinâmica	Gera uma solução resolvendo simples subproblemas.
Backtrack e *Branch Bound*	Encontra uma solução ótima para recursivamente dividir uma região em subdomínios.
Modelos de Grafos	Construção de grafos que representam variáveis aleatórias como nós e dependências condicionas como arestas.
Máquinas de Estados Finitos	Sistema de comportamento definido por estados, transições definidas por entradas e estados atuais associados com transições.

4 TRABALHOS RELACIONADOS

Muitos trabalhos seguem a mesma direção para a pesquisa de uma medida de desempenho confiável para aplicações científicas. Todos os trabalhos apresentados a seguir são baseados nesta pesquisa sobre a interação que existe entre as características das aplicações e o desempenho atingido, ambas para arquiteturas *multi-core* e *manycore* e ambientes virtualizados, especialmente para aqueles baseados em computação em nuvem. A caracterização dos *Dwarfs* é importante para este trabalho assim como o é para os trabalhos relacionados a seguir.

Desde que Phillip Colella, em sua apresentação de 2004 (COLELLA, 2004), disponibilizou a lista dos sete *Dwarfs* iniciais para categorizar os padrões de computação aplicados na computação científica, alguns pesquisadores vêm desenvolvendo melhorias e aplicando seus conceitos. Pesquisadores da Universidade da California, em Berkeley exploraram seu conceito e o estenderam para treze *Dwarfs* (ASANOVIC, 2006), com o interesse de aplicá-los para um maior número de métodos computacionais e investigar a qualidade de captura de padrões de computação e comunicação, para um maior número de tipos de aplicações científicas. Idealmente, os responsáveis por este traballho procuravam um indício de bom desempenho entre o conjunto dos *Dwarfs*, indicando que as novas arquiteturas *manycore* e alguns modelos de programação obteriam grande desempenho para um grande universo de aplicações futuras. Tradicionalmente, aplicações se baseiam em um HW e em modelos de programação existentes, mas ao invés disto, os autores procuravam o estudo desta classificação para o projeto de HW necessário para o futuro das aplicações científicas.

Em 2011, o trabalho de Springer et. al. (SPRINGER, 2011) foca em algumas implementações para GPU de alguns *Dwarfs* pré-selecionados e abre uma discussão sobre três suites de *benchmarks* que implementam um sub-conjunto dos treze *Dwarfs* na GPU. Neste trabalho são listados problemas típicos relacionados às implementações eficientes para aceleradores, discutindo sobre o desempenho e os problemas específicos quando esta classificação é aplicada às GPUs.

O projeto TORCH (KAISER, 2010) identifica vários *kernels* para propósitos relacionados à *benchmarking*, quando aplicados no contexto de HPC. Neste trabalho há a

argumentação de que um número de *benchmarks* existentes podem ser englobados como implementações de referência para um ou mais *kernels* disponíveis do TORCH. Estes *kernels* são classificados de acordo com os treze *Dwarfs* e os autores abrem a discussão para possíveis estratégias de otimizações de código que podem ser aplicadas neste contexto. Para cada *Dwarf*, alguns algoritmos são incluídos em sua suite, onde diferenciam-se em detalhes de implementação, ao mesmo tempo em que fazem parte de uma mesma classe de *Dwarf*.

No projeto *"The Parallel Dwarfs"* (SAX, 2013) também é adotada a classificação dos treze *Dwarfs* de Berkeley, usada para descrever a computação fundamental contida em seus *benchmarks*. Estes correspondem a uma suite de treze *kernels* paralelizados utilizando implementações em OpenMP, TPL e MPI.

As suites de *benchmarks* de código aberto Rodinia (CHE, 2009) e Parboil (STRATTON, 2012) implementam algoritmos de aplicações científicas mapeados no conjunto dos treze *Dwarfs*. As aplicações contidas no Rodinia são designados para infraestruturas de computação heterogêneas e usam OpenMP, OpenCL e CUDA para permitir comparações entre arquiteturas *multi-core* e *manycore*. Na suite Parboil estão contidas implementações completas e otimizadas para GPUs e algumas básicas para CPUs.

Alguns trabalhos propõem a caracterização das aplicações científicas para a melhora do desempenho em ambientes de computação em nuvem. A proposta de Muraleedharan (MURALEEDHARAN, 2012) é criar uma ferramenta (Hawk-i) para investigar como diferentes algoritmos científicos conseguem escalar em diferentes instâncias de nuvem da Amazon EC2 (AMAZON, 2013). Os experimentos contidos neste trabalho utilizam duas classes de *Dwarfs* e conduzem a um estudo de diferentes instâncias, utilizando a ajuda do Hawk-i para identificar instabilidades de desempenho em cada uma destas instâncias.

A investigação feita por Phillips et. al. (PHILLIPS, 2011) é baseada no desafio de determinar, qual provedor de IaaS e quais recursos são as melhores escolhas para executar aplicações que possuam requisitos específicos de QoS (*Quality of Service*). Para os autores, a ideia é obter a habilidade de prever o desempenho alcançado por uma aplicação através da disponibilização da descrição geral do HW usado pelo provedor de IaaS. Os *Dwarfs* foram utilizados para medir o desempenho do HW virtualizado, conduzindo experimentos nos serviços BonFIRE (BONFIRE, 2013) e Amazon EC2 (AMAZON, 2013). A partir disto, são mostradas quais diferentes combinações de HW devem ser utilizados para os diferentes tipos de computação. Além disto é demonstrado como o desempenho das

aplicações variam de acordo com o HW do provedor de nuvem. Isto pode ser refletido por *Dwarfs* que são mais sensíveis às diferenças entre as arquiteturas de seus hospedeiros do que à camada de virtualização aplicada, mesmo que por vezes as diferenças são pequenas, ou não demonstradas neste trabalho. Examinando as relações entre as diferentes classes, é demonstrado que os *Dwarfs* possuem a capacidade de medir diferentes aspectos de desempenho contidos no HW. A relação *entre* Dwarfs e aplicações científicas sugere que os estes podem ser úteis na previsão de desempenho de uma aplicação.

O trabalho de Engen et. al. (ENGEN, 2012) realiza a investigação do uso dos *Dwarfs* para realizar a caracterização dos recursos computacionais, que são baseados em entradas para um modelo de aplicações usado para prever o desempenho das aplicações em ambientes de nuvem. Baseados em investigações realizadas no BonFIRE e em outras nuvens públicas, os autores demonstram que a caracterização de recursos computacionais usando *Dwarfs* é feita com sucesso para a modelagem de aplicações para prever o desempenho de duas aplicações multimídia e uma científica.

Todos os trabalhos citados possuem semelhanças com o trabalho proposto nesta dissertação, principalmente no sentido de se basear na importância da caracterização de aplicações científicas ou de um melhor entendimento da infraestrutura disponível. Em alguns trabalhos são mapeados um conjunto de *Dwarfs* para serem usados em *benchmarks* ou *kernels*. Em (ASANOVIC, 2006) isto é feito isto para o SPEC2006 e EEMBC. Em (CHE, 2009) isto é feito para alguns *kernels* especializados em CPU e GPU e em (STRATTON, 2012) são mapeados exclusivamente para GPU. A ideia essencial destes trabalhos é a de que suas implementações servem como grandes estratégias em como avaliar uma determinada arquitetura. Contanto, não são oferecidos detalhes de implementação, nem como isso pode ser feito, além de não apresentarem resultados de experimentos destes ambientes virtuais a nível de hipervisor. Somente em (SPRINGER, 2011) que é realmente apresentado o comportamento de uma arquitetura de GPU para alguns *Dwarfs*. Em (MURALEEDHARAN, 2012) os autores propõem o uso dos *Dwarfs* para avaliar ambientes de nuvem, embora tenham somente avaliado seus comportamentos, sem compará-los com outras arquiteturas reais ou virtuais, o que os autores identificam como uma pesquisa necessária a ser feita. O mesmo ocorre nos trabalhos (PHILLIPS, 2011) e (ENGEN, 2012), onde os *Dwarfs* são usados para prever o desempenho para aplicações somente em ambientes de nuvem.

Enquanto alguns dos trabalhos citados focam exclusivamente em arquiteturas *many-*

core, *multi-core* ou em ambientes virtualizados, este trabalho propõe como contribuição um maior e mais completo ambiente de experimentos, onde serão avaliados CPUs *multi-core* e aceleradores *manycore* em ambientes reais e virtuais. Neste trabalho é explorado como as diferentes arquiteturas podem afetar o desempenho dos *Dwarfs*, além de identificar quais aspectos de configuração podem limitar seu desempenho em ambientes reais e virtuais. Isto é feito através da análise do impacto do hipervisor quando responsável por gerenciar um ambiente que execute um *Dwarf*. Neste trabalho são mostrados resultados de experimentos com a classe de *Dwarfs* DLA (*Dense Linear Algebra* - Álgebra Linear Densa), que é amplamente utilizada em aplicações científicas. No próximo capítulo a análise comparativa proposta neste trabalho é detalhada.

5 DESCRIÇÃO DA ANÁLISE COMPARATIVA PROPOSTA

Neste trabalho são apresentados quatro experimentos utilizando a classe de *Dwarfs* DLA. O algoritmo de DLA utilizado foi o LUD (*LU Decomposition* - Decomposição LU), disponível na suite Rodinia. LUD é um algoritmo usado para calcular soluções de um conjunto de equações lineares. O *kernel* LUD decompõe uma matriz como o produto da matriz triangular superior e da matriz triangular inferior e sua solução é dada através de um algoritmo intensivo de processamento. O objetivo destes experimentos é explorar o comportamento dos aceleradores (GPU e Xeon Phi™) e de uma CPU disposta de um número de núcleos próximo ao número do Xeon Phi™, estabelecendo um comparativo entre eles. Além disto, foi explorado o efeito que a camada de virtualização (KVM e XEN) exerce frente à execução de aplicações científicas com tipos diferentes e implementações (CUDA e OpenCL).

5.1 DWARF UTILIZADO E SUITE RODINIA

Neste trabalho é utilizado o *Dwarf* DLA. As motivações para a escolha deste *Dwarf* como o primeiro a ser analisado se dá pelos diversos domínios (classificados como DLA) que estão presentes em aplicações científicas. Como este *Dwarf* é caracterizado pela sua intensidade de processamento, há uma grande capacidade de paralelização deste tipo de problema, permitindo a análise dos experimentos utilizando um maior número de arquiteturas paralelas.

Por conseguinte, este é o *Dwarf* que melhor se adequa para o uso em arquiteturas *multi-core* e *manycore*, sendo um dos *Dwarfs* com um maior número de implementações disponibilizadas (SPRINGER, 2011). Estas características estão alinhadas com o propósito deste trabalho, motivando assim, a sua escolha.

Neste trabalho será utilizado a suite de *benchmarks* Rodinia, que é voltada para a computação paralela e heterogênea, sendo baseada nos *Dwarfs*. A principal razão da escolha do Rodinia é pela disponibilização de diversos *benchmarks* em variadas áreas da computação, tais como: geração de imagens médicas, bioinformática, dinâmica de fluidos, álgebra linear, simulações físicas, reconhecimento de padrões, mineração de dados, entre outros, todos direcionados à abordagem dos *Dwarfs*.

A suite Rodinia também oferece a disponibilização dos códigos em diversas linguagens, com isso, os *benchmarks* que compõem este suite podem ser executados tanto em CPU (com o OpenMP ou OpenCL) quanto em GPU (através do CUDA ou OpenCL). Como a finalidade deste trabalho é definir o atual estado da arte da virtualização de GPUs e poder englobar também a execução em arquiteturas dos coprocessadores Intel Xeon Phi, o Rodinia se faz a suite de *benchmarks* mais adequada.

Na Seção seguinte são descritas as especificações de arquitetura utilizada neste trabalho.

5.2 ARQUITETURAS UTILIZADAS

- Arquitetura *multi-core* CPU AMD Opteron 6376

  A arquitetura de CPU que compõe o ambiente *multi-core* utilizado neste trabalho é um sistema com 64 núcleos reais, divididos em 4 processadores AMD Opteron 6376 (AMD, 2013), com 16 núcleos cada e operando a 2.3 GHz, com um total de 128 GB de memória RAM instalada. Cada processador é dividido em 2 bancos de 8 núcleos cada, cada um com sua própria memória cache L3 de 8MB. Cada banco é então dividido em sub-conjuntos de 2 núcleos, compartilhando 2MB de cache L2 e 64KB de cache de instruções. Cada núcleo tem sua própria memória cache L1 de 16KB. A escolha desta arquitetura se teve pelo seu número de núcleos, que se equipara à quantidade de núcleos reais do acelerador Intel Xeon Phi.

- Acelerador *manycore* Intel® Xeon Phi™ 5110P (acelerador x86)

  O acelerador Xeon Phi™ utilizado neste trabalho é disposto de 60 núcleos reais. Cada núcleo suporta 4 HW threads. A arquitetura se baseia em núcleos x86 com unidades de vetorização de 512 bits, executando-os a aproximadamente 1GHz, chegando ao pico teórico de desempenho de 1 TeraFlop (TF) com precisão dupla, além de possuir um SO Linux customizado. O Xeon Phi™ também possui 8GB de memória RAM GDDR5. Como cada núcleo do acelerador é conectado a uma memória cache própria, o processador pode minimizar a perda de desempenho que pode ocorrer se cada núcleo recorre à memória RAM constantemente. Como o Xeon Phi™ possui seu próprio SO e é capaz de executar os códigos nativamente, ele foi utilizado como um nó de execução independente, compilando o código no seu

hospedeiro e transferindo os executáveis para o acelerador.

- Acelerador *manycore* GPU Nvidia Tesla M2050 (acelerador GPU)
  O acelerador baseado em GPU usado neste trabalho foi a Nvidia Tesla M2050, dedicada exclusivamente para o processamento de aplicações cientificas, sem saída gráfica. Esta GPU trabalha a 1.15 GHz no clock do processador e a 1.54GHz no clock da memória, possui 3GB de memória RAM GDDR5 e 148GB/seg de largura de banda de memória. Este acelerador tem um pico teórico de desempenho de 1.03 TF de operações de precisão simples e 515 GF com operações de precisão dupla.

5.3 DESCRIÇÃO DOS EXPERIMENTOS

Nos experimentos contidos neste trabalho foram utilizadas as implementações padrões contidas no Rodinia e nenhuma configuração especial foi feita para executar nas CPUs e nos aceleradores utilizados neste trabalho. Isto foi feito com a intenção da eficácia da comparação entre os mesmos códigos, excluindo diferenças que possam ocorrer com otimizações pontuais em cada plataforma de execução. As alterações feitas em cada implementação foram referentes ao uso de estratégias de vetorização, que compõem partes críticas dos códigos a serem corretamente paralelizadas em cada arquitetura.

Para cada resultado exposto em cada experimento, foram feitas 30 execuções e a partir disto, foram calculados os tempos médios para cada teste. Os intervalos de confiança para os testes foram menores que 1%, logo, não são mostrados nos gráficos.

Os tamanhos das matrizes de entrada foram limitados para assegurar que o espaço total de memória necessária para alocar a matriz estivesse disponível na memória de todos os aceleradores utilizados nos testes. Os tamanhos das matrizes de entrada foram definidos como matrizes quadradas de: 1024, 2048, 4096, 8192 e 16384 elementos de ponto flutuante com precisão simples.

Na subseção seguinte são mostradas as especificações de cada experimento:

5.3.1 EXPERIMENTOS NO AMBIENTE REAL

No ambiente real, o principal objetivo é medir o desempenho alcançado pela utilização de arquiteturas padrão x86 (CPU multi-core e Xeon Phi™), além de estabelecer um comparativo entre estas arquiteturas e um acelerador baseado em GPU. A intenção é definir qual arquitetura melhor se adequa ao algoritmo LUD, utilizando a mesma estratégia para todas arquiteturas, como feito em (CAO, 2013) e (DOLBEAU, 2013). Nestes testes foram utilizadas as versões padrões das implementações do LUD em OpenMP e OpenCL, contidas no Rodinia.

Para extender a capacidade de um ambiente multi-processado, o número de *threads* para cada teste foi especificado para alocar todos os núcleos disponíveis. Além disto, foi especificada a afinidade *"scatter"* , como a melhor afinidade entre threads disponível para este tipo de problema (S. CADAMBI, 2013).

Para alcançar um cenário que seja possível definir uma comparação entre as arquitetura x86 (CPU e Xeon Phi™), foi definida uma porcentagem de uso para os núcleos disponívies (25%, 50%, 75% e 100%). Na prática, isto significa que um sistema com 64 núcleos disponíveis irá executar com 50% de sua carga se 32 núcleos estão sendo utilizados pela aplicação. A mesma relação pode ser feita com o acelerador Xeon Phi™ e seus núcleos disponíveis.

O primeiro experimento neste ambiente foi definido para avaliar as arquiteturas x86 presentes neste trabalho (CPU e acelerador Xeon Phi™), executando a implementação em OpenMP do algoritmo LUD. Para estes testes, foram utilizadas as arquitetura de CPUs *multi-core* detalhada anteriormente e o acelerador x86 Intel Xeon Phi™, que foi configurado para operar como um nó (SMP) em separado. Como um nó de computação, este acelerador pode executar aplicações sem a intervenção do processador do seu hospedeiro, trabalhando somente com sua própria memória GDDR5. Para isto, a aplicação deve ser compilada no hospedeiro com os compiladores próprios da Intel, e o arquivo executável deve ser enviado ao acelerador, possibilitando a sua execução nativa. Este experimento pretende avaliar o comportamento do problema LUD, implementado em OpenMP, executando em arquiteturas *multi-core* e *manycore* baseadas no mesmo modelo de arquiteturas de processadores (x86).

No segundo experimento foi possível incorporar um segundo tipo de acelerador (baseado em GPU) aos testes. Para isso, foi usada a implementação em OpenCL do LUD, com a finalidade de avaliar como cada arquitetura se comporta na execução do mesmo pro-

blema, com a mesma implementação. Neste experimento a aplicação foi executada na CPU *multi-core* x86, no acelerador x86 (Xeon Phi™) e no acelerador baseado em GPU. O tempo considerado para a avalização dos experimentos foi o tempo necessário para envio da informação, processar e receber a informação resultante (*Wall time*).

5.3.2 EXPERIMENTOS NO AMBIENTE VIRTUAL

No ambiente virtual, o primeiro experimento foi feito com a intenção de avaliar o comportamento e o impacto da camada de virtualização (hipervisor) no mesmo algoritmo LUD, implementado em CUDA. No segundo experimento, a intenção é avaliar o mesmo comportamento com uma implementação heterogênea em OpenCL. Os resultados dos experimentos no ambiente virtual oferecem uma base de referência para:

- Investigar o atual estágio das tecnologias de *PCI passthrough* e IOMMU;
- Verificar o comportamento dos resultados alcançados com cada linguagem de programação;
- Analisar se a estratégia de implementação modifica a maneira de como o hipervisor trata o acesso ao dispositivo, quando executando o *kernel* LUD.

.

Em ambos experimentos neste ambiente, somente os *kernels* de processamento são enviados ao acelerador, o que se deve às linguagens utilizadas neste experimento (CUDA e OpenCL), que executam como um programa em *"offload"* , enviando somente a parte intensiva de processamento ao acelerador. Em adição a isto, as MVs utilizadas executaram no mesmo hospederio (sem concorrência), com a mesma configuração (10GB de memória RAM, 10 núcleos e 50GB disponívieis para sistemas de arquivos), utilizando o mesmo processador do servidor hospedeiro.

No próximo capítulo são apresentados os resultados e a análise realizada com base nos experimentos aqui descritos.

6 ANÁLISE DOS RESULTADOS

Como mencionado anteriormente, os experimentos foram submetidos aos ambientes real e virtual. Nas seções seguintes são mostrados os resultados, o comportamento e a investigação feita em cada ambiente.

6.1 RESULTADOS NO AMBIENTE REAL

Inicialmente foram feitos os experimentos comparando as arquiteturas CPU multi-core e o Xeon Phi™. As figuras 6.1, 6.2 e 6.3 mostram a relação entre o tempo de execução (em segundos) e a porcentagem de utilização dos núcleos. Neste primeiro experimento somente são mostrados os gráficos para os tamanhos de matrizes quadradas de 1024, 4096 e 16384 de matrizes de entrada, pois são os casos onde as mudanças significantes se encontram com mais frequência.

No primeiro experimento no ambiente real, pode-se notar que utilizando uma matriz quadrada de 1024 (figura 6.1) como entrada, o tempo de alocação dos núcleos produz um impacto maior que o tempo de processamento. Para 25% de alocação dos núcleos, a CPU multi-core x86 gasta 20% mais tempo tempo do que o acelerador baseado em x86, o que se deve à maior demora na alocação dos núcleos dentro do acelerador, o que gera pouco impacto no tempo de execução (*Wall time*) com poucos núcleos sendo utilizados.

A partir do aumento da utilização dos núcleos disponívieis (50% ou mais), o tempo de alocação para o acelerador baseado em x86 gera um maior impacto no tempo total de execução, ocasionando uma perda de até 70%, no pior caso. A principal característica deste comportamento é o pequeno tamanho da matriz de entrada, juntamente à grande quantidade de núcleos e *hardware threads* presentes no acelerador utilizados neste experimento, o que ocasionou em pouco tempo dedicado exclusivamente ao processamento.

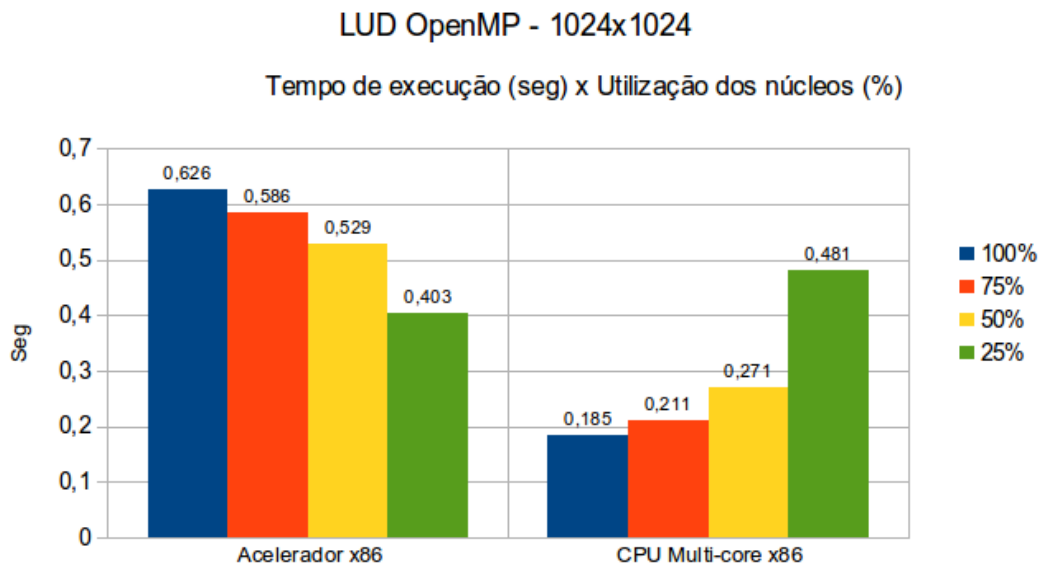

FIG. 6.1: Teste de desempenho com matriz quadrada 1024 em OpenMP.

Quando o tamanho da matriz quadrada é aumentado para 4096 e 16384, os resultados mostram que o tempo de alocação para o acelerador x86 acarreta em menos impacto no tempo total de execução do problema, gastando até 81% menos tempo para o caso de matrizes quadradas de 4096 (figura 6.2) e chegando a utilizar até 65% menos tempo para as matrizes quadradas de 16384 (figura 6.3), como melhores casos. Na figura 6.2 também pode-se notar que, embora o acelerador x86 tenha alcançado melhor desempenho, ele não foi capaz de atingir uma boa taxa de escalabilidade para mais de 50% da alocação dos seus núcleos. A razão para isso é o impacto do tempo de alocação dos núcleos, reduzindo o ganho obtido com a capacidade de processamento, o que é agravado pelo tamanho de matriz de entrada (ainda considerado pequeno para uso otimizado do acelerador x86).

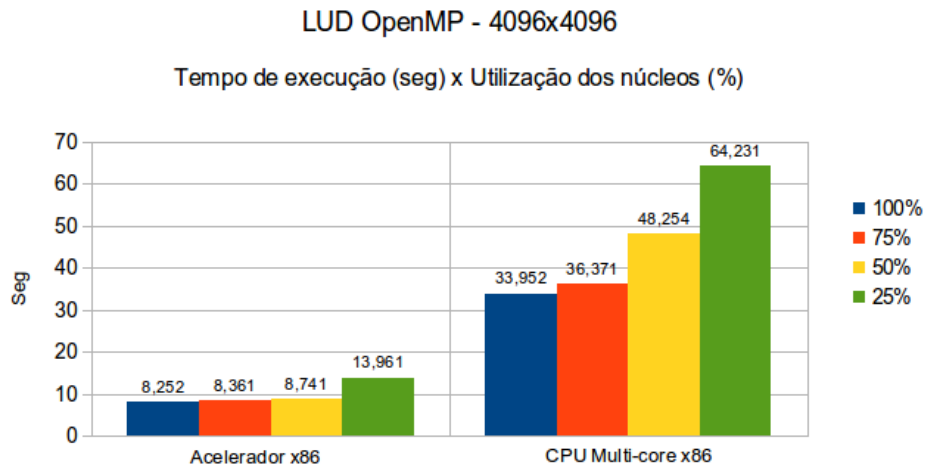

FIG. 6.2: Teste de desempenho com matriz quadrada 4096 em OpenMP.

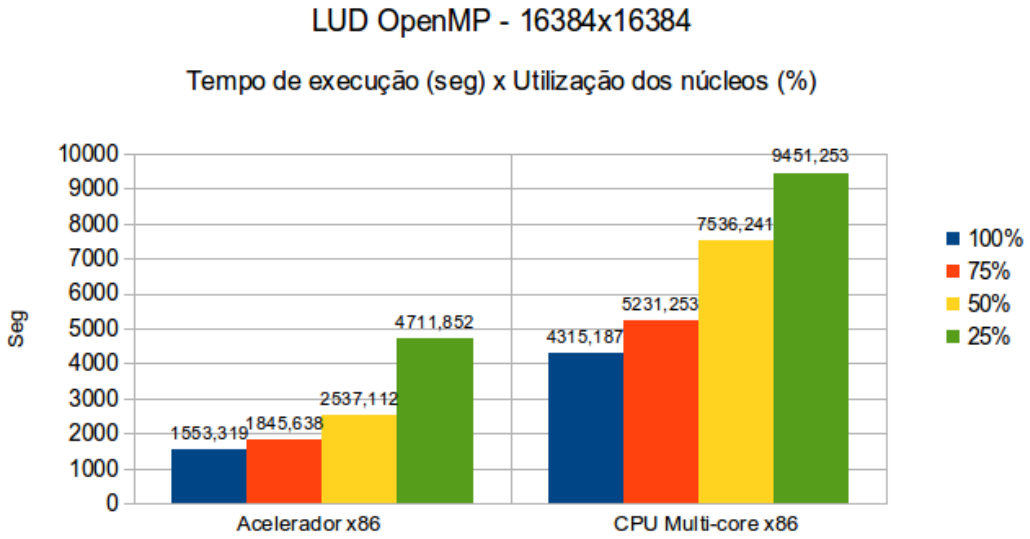

FIG. 6.3: Teste de desempenho com matriz quadrada 16384 em OpenMP.

O segundo experimento submetido a este ambiente (figura 6.4) mostra a relação entre o tempo de execução (em milissegundos) e os tamanhos das matrizes de entrada para a CPU multi-core baseada em x86, o acelerador baseado em x86 e o acelerador baseado em GPU. Para destacar as diferenças obtidas, optou-se por usar a escala logarítmica no eixo das ordenadas. Da mesma maneira que os gráficos anteriormente dispostos, para comparar as arquiteturas é necessário focar a comparação nas colunas com os mesmos

tamanhos de matrizes.

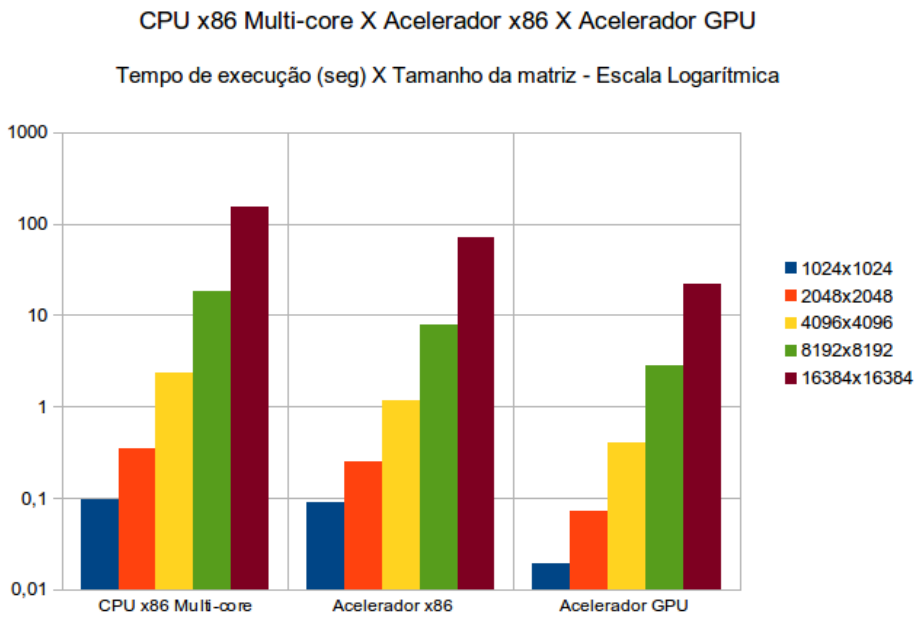

FIG. 6.4: Teste de desempenho com todas as arquiteturas em OpenCL.

A mesma implementação em OpenCL foi submetida em todas arquiteturas utilizadas neste trabalho. É possível verificar que o acelerador baseado em GPU obteve os melhores resultados. Isto se deve ao número de núcleos massivamente paralelos e à capacidade de processamento alcançada através deste paralelismo, o que sobrepõe ao efeito de transferir os dados ao acelerador. Um exemplo disto, é a coluna da matriz quadrada de 1024 elementos, onde as arquiteturas baseadas em x86 (CPU e acelerador) obtiveram tempos de execução seme-lhantes, enquanto o acelerador baseado em GPU foi cerca de quatro vezes mais rápido. Com o aumento do tamanhos das matrizes de entrada, o acelerador baseado em GPU obteve sempre os melhores resultados, um exemplo é o caso da matriz quadrada de 16384, onde executou cerca de sete vezes mais rápido que a CPU multi-core e três vezes mais rápido que o acelerador x86, o que é exibido na tabela 6.1. Neste caso, para estes tamanhos de matrizes, para o problema LUD, caracterizado como um *Dwarf* intensivo de processamento e implementado em OpenCL, o desempenho alcançado pela GPU utilizada foi melhor do que as demais arquiteturas analisadas neste trabalho.

TAB. 6.1: Tempos médios de execução em OpenCL para todas arquiteturas.

Tamanho das matrizes X Arquitetura utilizada	Tempos de execução (ms)		
	CPU x86 multi-core	**Acelerador x86**	**Acelerador GPU**
1024x1024	96,692	88,108	19,126
2048x2048	346,88	254,221	71,231
4096x4096	2372,749	1152,654	406,737
8192x8182	18297,384	7900,895	2840,296
16384x16384	151783,944	71340,485	22086,839

6.2 RESULTADOS NO AMBIENTE VIRTUAL

Em ambos experimentos realizados nos ambientes virtuais (implementações do algoritmo LUD em OpenCL e em CUDA) pode ser notado que para todos os tamanhos de matrizes de entrada, os ambientes real e virtual permaneceram com níveis semelhantes de tempos de execução (figuras 6.5 e 6.10). Isto se deve à maturidade das tecnologias de *PCI passthrough* usada em ambos os hipervisores e IOMMU implementado no *hardware* usado neste trabalho.

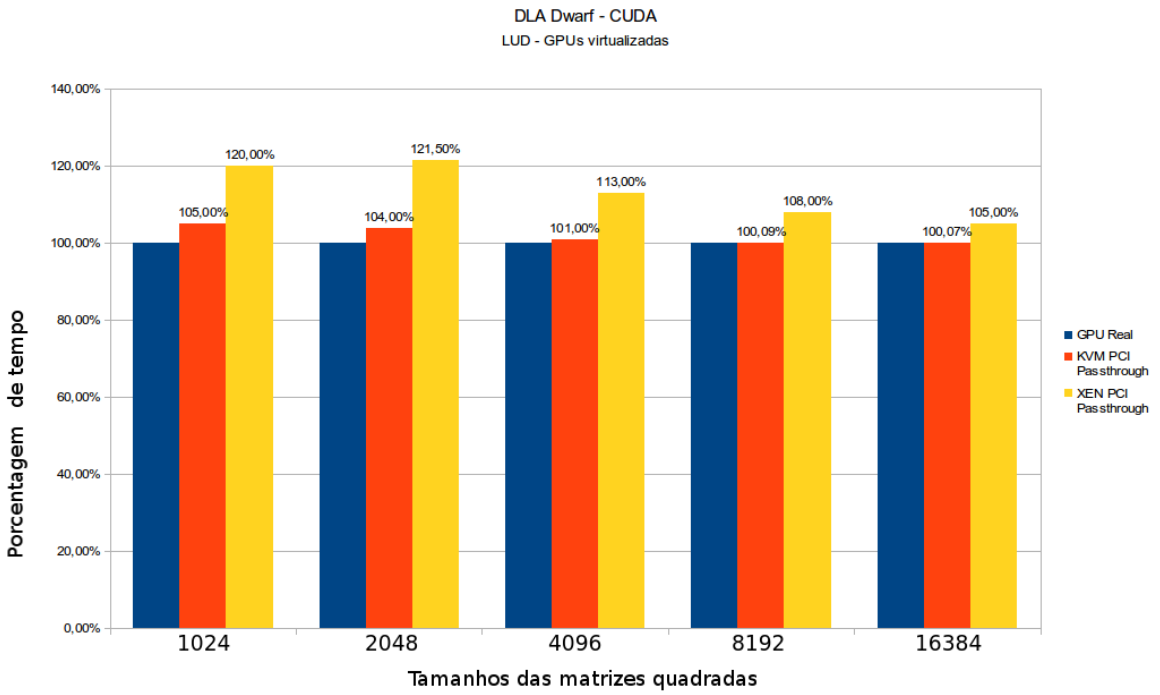

FIG. 6.5: Comparativo de desempenho em CUDA nos ambientes virtuais X ambiente real.

Os desempenhos aproximados são alcançados principalmente pelo emulador QEMU utilizado em ambos hipervisores e customizado de acordo com as características próprias de cada um, refletindo diretamente na maneira com que cada Sistema Operacional convidado trata o acesso aos dispositivos PCI.

No primeiro experimento para o ambiente virtual, foi utilizada a implementação em CUDA do algoritmo LUD, previamente selecionado. Sendo executado no sistema hospedeiro nativamente (sem a camada de virtualização), em uma máquina virtual criada com o KVM e em uma máquina virtual criada com o XEN. Para uma melhor visualização dos gráficos a seguir, foi previamente definido que a coluna referente à execução no hospedeiro seria exposta como 100% do tempo utilizado, com isso, as colunas que se referem aos hipervisores (KVM e XEN) estabelecem uma comparação de porcentagem diretamente ligada à execução no hospedeiro, demonstrando a relação de perda entre estes.

Nos resultados para submissão dos experimentos com a implementação em CUDA (figura 6.5), tem-se que em todos os casos a execução no hospedeiro (sem a camada de virtualização) foi a mais rápida.

Para o hipervisor KVM, o pior caso ocorreu com o uso das matrizes 1024x1024, onde

o uso da camada de virtualização acarretou em uma perda de 5% comparado à execução nativa no hospedeiro. Além disso, foi percebido que o melhor caso de execução foi com o uso da matriz quadrada de 16384 elementos, onde notou-se uma perda praticamente nula, com menos de 1% se comparado à execução nativa.

Estes resultados demonstram que o KVM implementa com maturidade a tecnologia de *PCI passthrough*, com seus tempos de execução muito próximos aos tempos alcançados do acesso direto do hospedeiro ao dispositivo, sendo uma boa escolha como estratégia de virtualização em ambientes que possam utilizar periféricos de E/S dedicados exclusivamente a uma máquina virtual. Uma das grandes contribuições para estes resultados é a proximidade do KVM com o kernel do Sistema Operacional. Atualmente, as bibliotecas padrões deste hipervisor são integradas por padrão no kernel do Linux, com isso, são utilizadas ferramentas nativas do kernel como controle dos dispositivos PCI pelos ambientes virtuais. Esta situação não acontece em um ambiente virtualizado com o XEN, onde é necessário implementar soluções diferentes para o controle dos mesmos dispositivos em um domínio criado com o XEN, como demonstrado a seguir.

Pode-se notar que o hipervisor que obteve menor desempenho neste *Dwarf* intensivo de processamento foi o XEN.

Comparando os tempos obtidos com o XEN à execução no Hospedeiro sem a camada de virtualização, pode-se notar que as taxas de perda para este hipervisor foram sempre maiores, chegando aos 21,5% para a matriz quadrada de 2048. Conforme o tamanho da entrada aumenta, esta taxa diminui, sendo possível verificar que, no caso para a matriz quadrada de 16384, a perda com o uso do XEN se estabeleceu em 5% comparado ao hospededeiro.

Comparando os tempos obtidos entre os hipervisores, tem-se que o XEN apresentou menor desempenho, com o pior caso para a matriz quadrada de 2048, consumindo 17,5% mais tempo do que o KVM. Isto se deve à implementação do XEN na alteração do emulador de *hardware* QEMU (utilizado em ambos virtualizadores). O XEN implementa o que é chamado de *"Stub Domain"* ou *"Driver Domain"*, considerado um serviço do XEN ou um domínio próprio para execução de instruções referentes aos dispositivos que implementam o *PCI passthrough*, este novo domínio é utilizado para prover maior segurança no envio de informações para o dispositivo diretamente acessado pela máquina virtual, porém acarreta em uma sobrecarga nos tempos de envio/execução/recebimento das informações para o dispositivo.

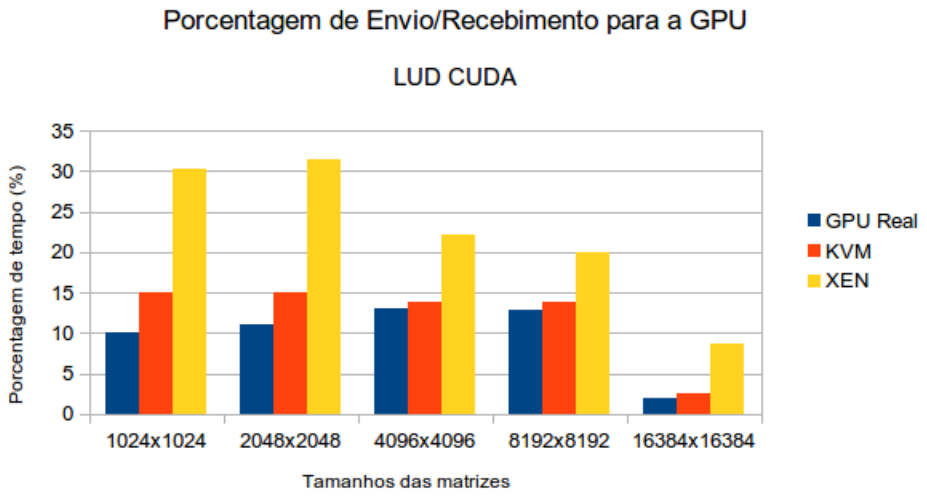

FIG. 6.6: Porcentagem de tempo para envio e recebimento.

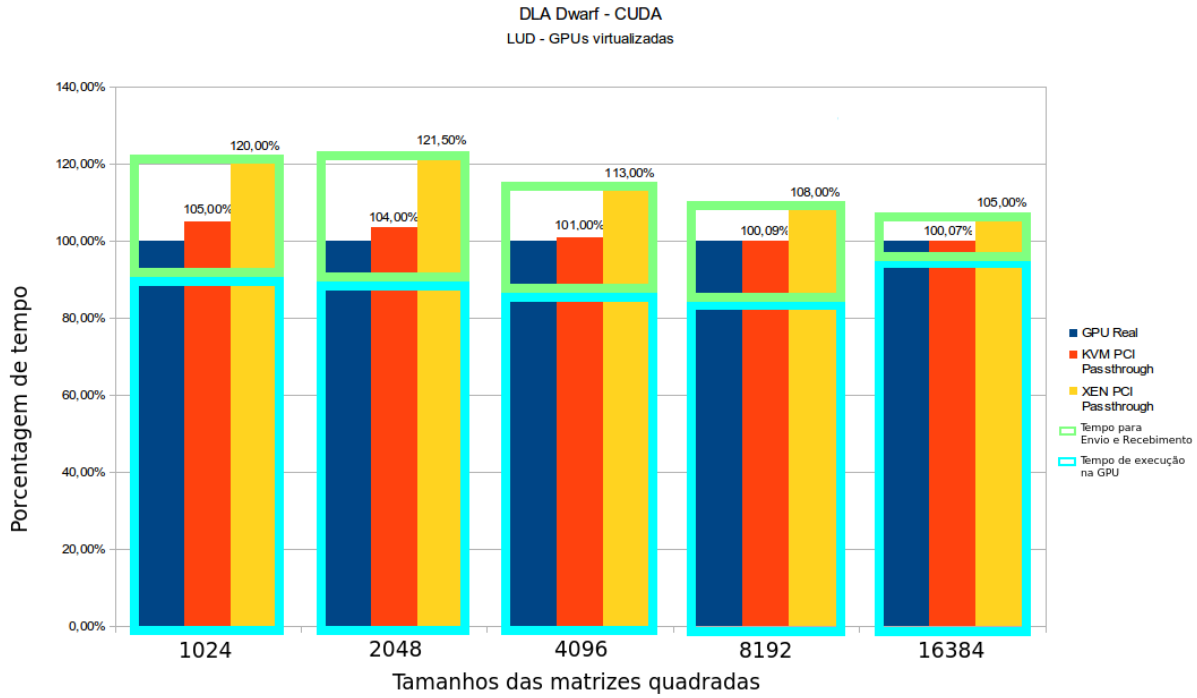

FIG. 6.7: Porcentagem de tempo para envio/recebimento e execução.

Nas figuras 6.6 e 6.7, pode-se notar que o grande problema no uso de um ambiente virtual com *PCI passthrough* é o envio e o recebimento de informações entre a máquina virtual e o dispositivo. Uma vez que os dados já estão dentro da GPU, o acelerador não necessita mais do hipervisor para auxiliar na execução. Além disto, pode-se notar que

o hipervisor XEN obteve os maiores tempos necessários para isto, permanecendo muito acima aos outros ambientes aqui testados, chegando a utilizar 32% do seu tempo para enviar e receber os dados a serem processados, enquanto os outros ambientes não passaram de 15% do tempo utilizados para esta finalidade. A figura 6.6 mostra as porcentagens dos tempos levando em consideração as porcentagens totais obtidas na figura 6.5.

No XEN, isto acontece pelo uso da chamada *Shadow Page Table*, que é um espelhamento da *Page Table*, cuja finalidade é permitir a integridade da localização dos dados em memória, fazendo com que o hipervisor tenha um controle de todas suas alterações. Este recurso é principalmente utilizado no processo de migração em tempo real das máquinas virtuais, porém acarreta em perda de desempenho no uso do *PCI Passthrough*.

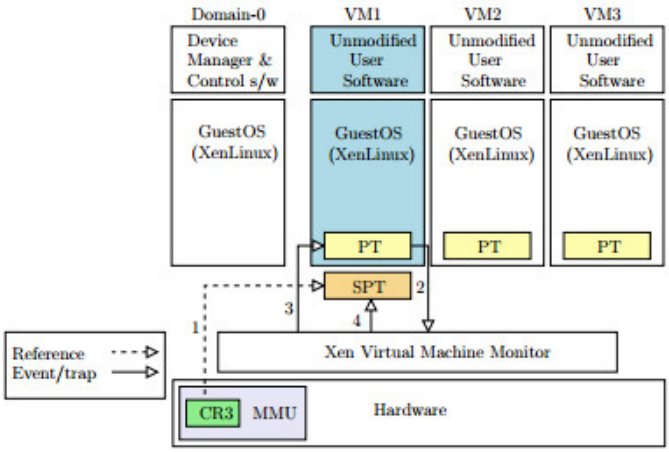

FIG. 6.8: Processo de gerenciamento de Shadow Page Table e Page Table pelo XEN.

Na figura 6.8, foi percebido que em modo normal de funcionamento, as mudanças são gerenciadas pela MMU, mas o XEN possui um registrador CR3 (que é responsável por traduzir endereços virtuais em endereços físicos, localizando o *page directory* e *page tables* para as aplicações em execução) que aponta para a *Shadow Page* que não é acessada pelo ambiente virtualizado. A VM pode acessar um espaço de endereçamento completamente virtualizado e somente pode mapear os limites alocados pelo hipervisor. Quando o SO do ambiente virtualizado tenta escrever na *Page Table*, esta ação é interceptada pelo hipervisor, que sempre fará a validação da operação antes de permitir a escrita na *Page Table*, garantindo o isolamento entre diversas VMs. Se a alteração for validada, ela será assim propagada para a *Shadow Page Table*.

No caso de um *Page Fault* no ambiente virtual, ele será interceptado pelo hipervisor (figura 6.9). Ao ser verificado que o erro é resultante da paginação da memória, será então devolvido ao ambiente virtual para que seja tratado e a *Page Table* é colocada em *read only*. O SO do ambiente virtual recupera a página, mas ao tentar escrever na *Page Table*, esta ação será novamente interceptada e validada pelo hipervisor, que então desbloqueará a *Page Table* (que estava em *read only* para a sincronização das páginas) para que possa ser escrita pelo ambiente virtual, e então o hipervisor atualizará a *Shadow Page Table*.

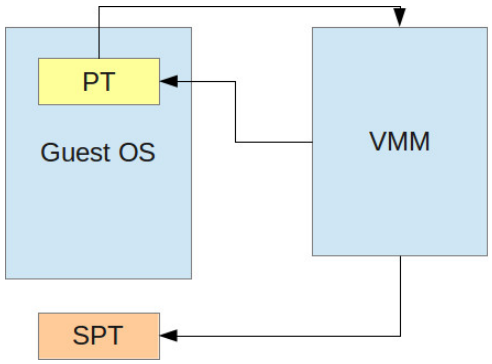

FIG. 6.9: Processo de tratamento de page fault no ambiente virtual pelo XEN.

Esta sobrecarga no tratamento dos dados manipulados pelo dispositivo foi o principal motivo de perda de desempenho do XEN.

No segundo experimento no ambiente virtual, a implementação em OpenCL do mesmo problema foi submetido ao mesmo ambiente. Como demonstrado na figura 6.10, mais uma vez o ambiente real obteve os melhores resultados dentre todos os ambientes.

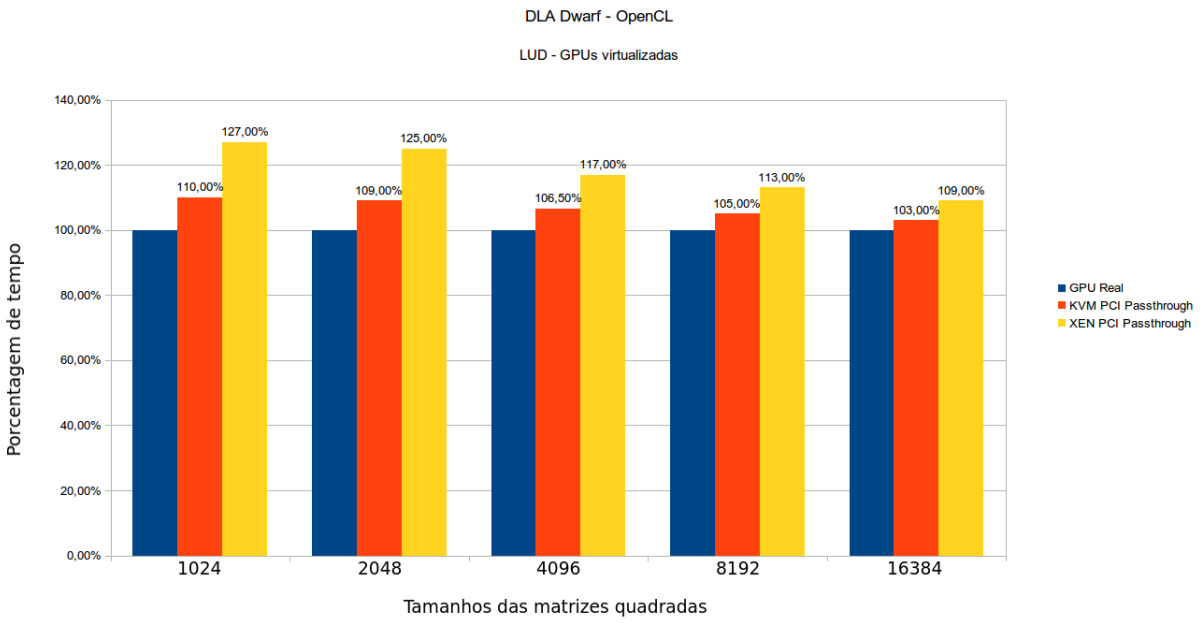

FIG. 6.10: Comparativo de performance em OpenCL nos ambientes virtuais X ambiente real.

Comparando os tempos obtidos com o ambiente virtual à execução no hospedeiro sem a camada de virtualização, foi percebido que as taxas de perda para ambos os hipervisores foram maiores do que as obtidas anteriormente com a implementação em CUDA. Podemos notar que o uso do KVM acarretou em perdas de até 10% (1024x1024), enquanto o XEN obteve uma perda de até 27% (1024x1024).

Comparando os tempos obtidos entre os hipervisores, temos que o XEN apresentou menor desempenho também para este experimento, chegando até 16% (matrizes quadradas de 1024 e 2048) comparado ao KVM. Neste experimento em OpenCL, foi notado que além do tratamento da *Shadow Page Table* pelo XEN, há também as diferenças entre as implementações em OpenCL e CUDA, que também acarretaram em taxas maiores de perda entre os ambientes virtuais e real. Estas diferenças são demonstradas nos trechos de códigos das Listagens 1 e 2.

Na listagem 1, tem-se um trecho do código em CUDA do problema utilizado neste trabalho. Pode-se verificar que a contagem do tempo de execução se dá antes da função de envio dos dados para a GPU e engloba o processamento e o recebimento dos dados processados. Após o trecho de envio, tem-se a chamada do controle dos kernels (detalhado a partir da linha 17 da listagem 1), seguido pela inha que termina a contagem do tempo (linha 13 da listagem 1).

No controle dos kernels, tem-se um laço de repetição o qual contém a principal diferença nos resultados obtidos, onde cada kernel (lud-diagonal, lud-perimeter e lud-internal) são mandados diretamente para a GPU processar. Cada kernel recebe seus argumentos para execução diretamente na sua chamada, sem a necessidade da intervenção de outras funções para auxiliar e definir a parte do problema a ser tratado em determinado laço de repetição, diferente do assumido no trecho de código em OpenCL.

```
1   /* Inicio da contagem do tempo de execucao */
2   stopwatch_start(&sw);
3
4   cudaMemcpy(d_m, m, matrix_dim*matrix_dim*sizeof(float),
5       cudaMemcpyHostToDevice); /* Envio das informacoes para a GPU */
6
7   lud_cuda(d_m, matrix_dim);    /* Chamada do Controle dos Kernels */
8
9   cudaMemcpy(m, d_m, matrix_dim*matrix_dim*sizeof(float),
10      cudaMemcpyDeviceToHost); /* Recebimento das informacoes processadas
            pela GPU */
11
12  /* Fim da contagem do tempo de execucao */
13  stopwatch_stop(&sw);
14
15 ...
16
17 /* Controle dos Kernels */
18
19 void lud_cuda(float *m, int matrix_dim)
20 {
21   int i=0;
22   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
23   float *m_debug = (float*)malloc(matrix_dim*matrix_dim*sizeof(float));
24
25   for (i=0; i < matrix_dim-BLOCK_SIZE; i += BLOCK_SIZE)
26   {
27
28     lud_diagonal<<<1, BLOCK_SIZE>>>(m, matrix_dim, i);
29
30     lud_perimeter<<<(matrix_dim-i)/BLOCK_SIZE-1, BLOCK_SIZE*2>>>(m,
```

```
        matrix_dim, i);

    dim3 dimGrid((matrix_dim-i)/BLOCK_SIZE-1, (matrix_dim-i)/BLOCK_SIZE-1);

    lud_internal<<<dimGrid, dimBlock>>>(m, matrix_dim, i);
  }

  lud_diagonal<<<1,BLOCK_SIZE>>>(m, matrix_dim, i);
}
```

Listing 6.1: Trecho do código em CUDA

```
...
/* Inicio da contagem do tempo de execucao */
stopwatch_start(&sw);

/* Envio das informacoes para a GPU */
err = clEnqueueWriteBuffer(cmd_queue, d_m, 1, 0, matrix_dim*matrix_dim*
    sizeof(float), m, 0, 0, 0);

if(err != CL_SUCCESS) { printf("ERROR: clEnqueueWriteBuffer d_m (size:%d)
    => %d\n", matrix_dim*matrix_dim, err); return -1; }

int i=0;
for (i=0; i < matrix_dim-BLOCK_SIZE; i += BLOCK_SIZE)
{

/* Exemplo da chamada de 1 kernel */

 /********* Inicio da definicao dos argumentos do Kernel *********/
  clSetKernelArg(diagnal, 0, sizeof(void *), (void*) &d_m);
  clSetKernelArg(diagnal, 1, sizeof(float) * BLOCK_SIZE * BLOCK_SIZE, (
      void*)NULL );
  clSetKernelArg(diagnal, 2, sizeof(cl_int), (void*) &matrix_dim);
  clSetKernelArg(diagnal, 3, sizeof(cl_int), (void*) &i);

  size_t global_work1[3] = {BLOCK_SIZE, 1, 1};
```

```
23      size_t local_work1[3]  = {BLOCK_SIZE, 1, 1};
24      /********* Fim da definicao dos argumentos *********/
25
26
27       /********* Chamada da execucao do kernel *********/
28      err = clEnqueueNDRangeKernel(cmd_queue, diagnal, 2, NULL, global_work1,
            local_work1, 0, 0, 0);
29      if(err != CL_SUCCESS) { printf("ERROR:  diagnal clEnqueueNDRangeKernel
           ()=>%d failed\n", err); return -1; }
30
31    ...
32
33    }
34    /* Recebimento das informacoes processadas pela GPU */
35    err = clEnqueueReadBuffer(cmd_queue, d_m, 1, 0, matrix_dim*matrix_dim*
          sizeof(float), m, 0, 0, 0);
36
37    if(err != CL_SUCCESS) { printf("ERROR: clEnqueueReadBuffer  d_m (size:%d)
           => %d\n", matrix_dim*matrix_dim, err); return -1; }
38
39    clFinish(cmd_queue);
40
41    /* Fim da contagem do tempo de execucao */
42    stopwatch_stop(&sw);
43
44  }
```

Listing 6.2: Trecho do código em OpenCL

A listagem 2 exibe o trecho do código em OpenCL, o qual engloba o envio dos dados para a GPU (linha 6), o laço de repetição para controle e execução dos *kernels* (linha 13) e o recebimento das informações processadas pela GPU (linha 35), além do ponto de sincronização e espera de todas execuções (linha 39).

As grandes diferenças e onde se encontram as maiores taxas de perda dos resultados em OpenCL são resultadas da necessidade da chamada das funções *clSetKernelArg()* (linhas 17 à 20) e as definições de *globalwork* e *localwork* (linhas 22 e 23). Estas funções são responsáveis pela definição dos argumentos e do tamanho dos blocos que cada kernel irá

processar através da função *clEnqueueNDRangeKernel()*.

Como dito anteriormente, OpenCL é uma linguagem aberta para programação genérica para vários tipos de processadores, que cria uma interface de baixo nível. Como seu desenvolvimento teve como principal requisito o grande número de *devices* habilitados a executar um código em OpenCL, é necessário um controle mais complexo do ambiente em que determinada aplicação será executada. Para manter o controle da execução de vários *kernels* ao mesmo tempo em *devices* com arquiteturas diferentes, o OpenCL define um *context*, que é um conjunto de *devices, kernels* e objetos. A partir de um *context*, o *host* controla sua execução através de um objeto chamado *command-queue*. O *host* adiciona comandos a uma *command-queue*, que está associada a um *context*, e os comandos são executados dentro dos *devices*. Esta estratégia traz a necessidade de um maior controle sobre todos argumentos necessários para executar um *kernel*, além das informações do tipo de arquitetura onde um *kernel* será executado, com isso é inevitavelmente necessário obter um maior controle e configuração deste ambiente, que pode ser considerado 'híbrido'. Sendo assim, pode-se notar que estas funções de definição de argumentos e controle da *command-queue* e do *context* geram um controle maior feito pela CPU. Como as instruções são enviadas de um ambiente virtual, temos que os hipervisores geram o *overhead* encontrado nos casos utilizando a versão em OpenCL do problema LUD.

Através disto, pode-se verificar que uma implementação em OpenCL é muito mais dependente da CPU, se comparada a uma implementação em CUDA. Uma implementação em CUDA depende exclusivamente em como cada *kernel* é executado dentro de uma GPU. Uma implementação em OpenCL depende muito mais da intervenção da CPU para a definição dos argumentos necessários para executar em algum *device*. Isto pode ser notado também nas colunas relacionadas ao KVM na figura 6.10, onde os tempos de execução também diferem. Mais uma vez, para todos os tamanhos de matrizes de entrada utilizados neste experimento, o hospedeiro sem o uso da camada de virtualização obteve o melhor desempenho, seguido pelo KVM e pelo XEN.

7 CONSIDERAÇÕES FINAIS

A proposta apresentada neste trabalho teve como objetivo uma análise comparativa entre os principais aceleradores disponíveis no atual mercado de HPC. Buscou-se sempre em analisar como os principais hipervisores de código aberto acessam estes dispositivos quando há a necessidade de criação de um ambiente de computação paralela operando acima de uma camada de virtualização. Por meio desta análise, é possível definir comportamentos de aplicações científicas em determinados ambientes reais e virtuais, auxiliando, dessa forma, alunos, pesquisadores e usuários na aquisição de quais recursos seriam os ideais para determinados tipos de aplicações. A análise demonstrou as taxas de perda/-ganho que podem ser atingidas, além de definir qual o tipo de camada de virtualização melhor se comporta quando operando com acessos diretos aos aceleradores, que são parte fundamental das arquiteturas atuais de HPC.

Os experimentos executados neste trabalho permitiram não apenas verificar a viabilidade da aquisição de novos recursos (Xeon Phi) para o ComCiDis, mas também possibilitaram aprofundar os conhecimentos dos fatores tecnológicos envolvidos na instalação, gerenciamento e utilização destes recursos para um ambiente de HPDC.

O estudo apresentado no capítulo 6 levantou uma análise comparativa entre os ambientes utilizados neste trabalho (real e virtual), onde a intenção foi comparar o que existe como atual estado da arte de aceleradores e hipervisores. Isto foi feito utilizando o *benchmark* Rodinia, que é baseado na classificação dos *Dwarf*s e oferece versões de implementação de algoritmos utilizando várias linguagens e APIs (C, CUDA, OpenMP e OpenCL). Foi escolhido o algoritmo LUD, classificado como um *Dwarf* de Álgebra Linear Densa (DLA), para os testes neste trabalho.

O primeiro experimento no ambiente real consistiu em analisar arquiteturas (CPU *multi-core* e acelerador *manycore* Xeon Phi) com o mesmo modelo de projeto (x86). Para isto, foi utilizada a versão em OpenMP do algoritmo LUD, utilizado em ambientes de memória compartilhada, isto foi possível visto que o Xeon Phi é tratado como um nó de execução, que trabalha com seus próprios recursos, independente do sistema hospedeiro. Para os tamanhos menores de matrizes de entrada, o tempo de alocação dos núcleos foi o principal agravante dos tempos alcançado pelo Xeon Phi. Para matrizes maiores, sua

arquitetura (capacidade de alocação de um alto número de *threads*, associados à coerência de cache e acesso otimizado à memória) foi o responsável pelo melhor desempenho, se comparado à arquitetura CPU *multi-core*.

O segundo experimento no ambiente real consistiu em analisar todas as arquiteturas de HPC disponíveis no ComCiDis (CPU *multi-core*, acelerador baseado em GPU e acelerador Xeon Phi). Como as estratégias para execução na GPU diferem das abordagens utilizadas nas outras arquiteturas, foi utilizada a versão em OpenCL do algoritmo LUD. O OpenCL é um *framework* para o desenvolvimento de programas heterogêneos, que são capazes de executar em CPUs, GPUs e aceleradores em geral.

A partir dos resultados alcançados com estes experimentos no ambiente real, pode-se concluir que a GPU obteve a melhor desempenho dentre as arquiteturas utilizadas. Isto ocorre devido ao seu maior número de núcleos de processamento massivamente paralelos, o que sobrepõe o efeito do tempo de transferir os dados ao acelerador. A GPU foi cerca de quatro vezes, ou 400% mais rápida para o primeiro tamanho de matriz neste experimento. Com o aumento do tamanhos das matrizes de entrada, obteve sempre os melhores resultados, um exemplo é o caso da matriz quadrada de 16384 elementos, onde executou cerca de sete vezes mais rápida que a CPU *multi-core* e cerca de três vezes mais rápida que o acelerador Xeon Phi. Com isso, o acelerador baseado em GPU obteve sempre o melhor tempo de execução em todos os experimentos. É importante observar que, para o desenvolvimento de uma aplicação baseada em GPU, o esforço do aprendizado das estratégias de programação em GPUs (CUDA ou OpenCL), podendo acarretar em um tempo maior de implementação do projeto.

Com os mesmos experimentos, foi notado que o acelerador Xeon Phi está localizado entre a facilidade de desenvolver ou portar uma aplicação paralela para seu ambiente e o ganho de desempenho alcançado por um acelerador baseado em GPU. A partir disto, é importante salientar que as versões em OpenMP e OpenCL do LUD não necessitaram ser significativamente modificadas para executar no Xeon Phi, foi necessário somente especificar *pragmas* em seu código e compilá-lo com as *flags* do compilador necessárias. Além disto, também é importante notar que para o maior tamanho do problema de entrada, o Xeon Phi obteve uma performance cerca de três vezes maior que a CPU *multi-core* em OpenMP, sem necessitar de alterações significativas em sua implementação.

No ambiente virtual, foram submetidos dois experimentos, onde buscaram avaliar como os hipervisores tratam o acesso direto à dispositivos de E/S. Com estes experimen-

tos foi possível identificar pontos fracos presentes na maneira em que cada camada de virtualização trata o acesso a estes dispositivos, além de verificar como a diferença entre as implementações podem acarretar em mais perdas nos ambientes virtuais.

No primeiro experimento no ambiente virtual, foi submetido o algoritmo LUD em CUDA para o ambiente real (hospedeiro sem virtualização), em uma MV virtualizada com o KVM e em uma MV virtualizada com o XEN. Foi notado que o acesso direto (*PCI passthrough*) não influencia em como um código é executado dentro da GPU, mas sim no acesso à memória da GPU, permanecendo neste ponto as principais diferenças entre os tempos de execução, onde o XEN obteve sempre os piores resultados (17,5% de perda no pior caso, comparado ao KVM), devido à maneira em que seu *"Device Domain"* opera.

No segundo experimento, foi submetido o algoritmo LUD em OpenCL para os mesmos ambientes descritos no experimento anterior. Neste caso, além da perda de desempenho devido ao envio e recebimento de informações, observa-se que a diferença de implementações também ocasionou em perdas significativas. Em OpenCL, por ser um *framework* que opera com vários tipos de dispositivos, há uma necessidade muito maior de controle dos seus *kernels* de execução, que precisam de muito mais intervenção da CPU. Como em uma MV, a CPU é demultiplexada para atender às várias requisições do ambiente real e virtual, a camada de virtualização ocasiona em uma perda maior para este tipo de implementação. Neste caso, foram encontradas taxas de perda superiores às obtidas em CUDA (10% de perda para o KVM e 27% para o XEN nos piores casos).

Através destes experimentos, pode ser notado que o algoritmo LUD, classificado como um *Dwarf* de DLA, pode ser migrado para um ambiente virtualizado, usado em uma nuvem computacional sem sofrer significativas perdas de desempenho quando comparado a um ambiente real, principalmente se utilizado o desenvolvimento em CUDA e o hipervisor KVM (perda de 5% no pior caso). Isto se deve principalmente ao tipo de aplicação sendo usada, neste caso, um *Dwarf* DLA, caracterizado como um *Dwarf* intensivo de processamento e aplicável à estratégia de programação contida nos aceleradores (CUDA e OpenCL); também podemos chegar a esta conclusão devido à maturidade das tecnologias que compõem a funcionalidade de *PCI passthrough* no XEN e no KVM, juntamente à tecnologia de IOMMU implementada nos *hardwares* presentes atualmente no mercado.

7.1 CONTRIBUIÇÕES

Este trabalho contribuiu diretamente para a análise do atual estado da arte da virtualização de GPUs e das arquiteturas *manycore* disponíveis atualmente, oferecendo informações do que pode ser esperado de ganho de desempenho quando da aquisição de novos equipamentos para um ambiente de HPDC.

Em consequência disto, contribuiu em como a classificação dos *Dwarfs* pode ser utilizada para avaliar uma arquitetura de processamento paralelo. Um melhor entendimento em como usá-los é muito importante para a área de HPC, pois nem todos centros científicos necessitam dispor de infraestruturas grandiosas, mas sim, identificar como cada arquitetura pode se comportar na execução de aplicações científicas, analisando qual o equipamento melhor se adequaria a sua necessidade.

Ao longo do desenvolvimento deste trabalho, foi estabelecido um contato com as empresas responsáveis pelo desenvolvimento das arquiteturas utilizadas. Através disto, foram identificadas possíveis melhorias nas estratégias de programação, configuração, uso e análise entre estas arquiteturas. Um exemplo disto é que a Intel disponibiliza os drivers necessários (*Manycore Platform Software Stack* - MPSS) para o acelerador Xeon Phi somente para versões do SO Linux empresariais (Red Hat Enterprise e SuSE Linux Enterprise Server), porém, no grupo ComCiDis somente são usados SOs Ubuntu Server. Em razão disto, foi necessário realizar análises e alterações nos códigos do MPSS e demonstrar à Intel que é necessário haver um *release* de versões do MPSS também para outras versões do Linux.

Em adição à estas contribuições, também foram levantados aspectos relevantes na configuração dos ambientes virtuais para a utilização do *PCI passthrough*. Para isto, foram gerados documentos técnicos para instalação dos ambientes virtuais, que oferecem um passo a passo para os requisitos, instalação e configuração necessários, auxiliando usuários, alunos e pesquisadores na implantação deste tipo de ambiente.

7.2 TRABALHOS FUTUROS

Como trabalhos futuros, pretende-se avaliar o comportamento das mesmas arquiteturas quando executam outras classes de *Dwarfs*. Por conseguinte, será avaliado o comportamento de aplicações específicas de grupos de pesquisas parceiros ao ComCiDis, que podem ser caracterizadas através dos *Dwarfs*. A principal motivação para isto é executar uma

análise semelhante à executada neste trabalho, buscando direcionar a melhor arquitetura para aplicações científicas específicas de determinadas áreas de pesquisa, auxiliando na melhor aquisição de recursos computacionais.

Além disto pode-se avaliar como cada hipervisor se comporta quando executam outros tipos de problemas e se há alguma alteração nos tempos de envio e recebimento de dados entre a MV e um dispositivo de E/S acessado diretamente com o *PCI passthrough*.

Para o algoritmo LUD (classe de *Dwarf* DLA), o tamanho da matriz de entrada será aumentado para avaliar como cada tipo de acelerador (GPU e Xeon Phi) lida com códigos em *offload*. Na GPU, a intenção é usar o LUD em CUDA, adicionando uma manipulação em como o envio é feito em cada acelerador enquanto este processa uma parte do problema (abordagem pipeline). Com isto seria possível também comparar implementações em OpenMP e em CUDA para identificar trechos que podem ser otimizados para cada arquitetura.

Em adição a isto, outros hipervisores de código aberto (com suporte ao *PCI passthrough*) podem ser avaliados, procurando prover uma lista completa de comportamento dos atuais hipervisores quando trabalham na gerência de MVs utilizadas na submissão de aplicações intensivas de diferentes métricas de desempenho.

8 REFERÊNCIAS BIBLIOGRÁFICAS

ADAMS e AGESEN. ***A comparison of software and hardware techniques for x86 virtualization.*** Em *In Proceedings of ASPLOS 06.*, 2006.

AMAZON. ***Amazon EC2 Instances***. `http://aws.amazon.com/hpc-applications/`, 2013. [Online; accessado em 19-Janeiro-2013].

AMD. ***AMD Processors for servers and Workstations***. `http://products.amd.com`, 2013. [Online; accessado em 10-Dezembro-2012].

ASANOVIC, K., BODIK, R., CATANZARO, B. C., GEBIS, J. J., HUSBANDS, P., KEUTZER, K., PATTERSON, D. A., PLISHKER, W. L., SHALF, J., WILLIAMS, S. W. e YELICK, K. A. ***the landscape of parallel computing research: a view from berkeley***. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

BLACKFORD, L. S., CHOI, J., CLEARY, A. J., DEMMEL, J., DHILLON, I. S., DONGARRA, J., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D. W. e WHALEY, R. C. ***ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance***. Em *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, pág. 5. IEEE, 1996.

BONFIRE, B. ***BonFIRE CLOUD***. `http://www.bonfire-project.eu/`, 2013. [Online; accessado em 18-Janeiro-2013].

CAO, C., DONGARRA, J., DU, P., GATES, M., LUSZCZEK, P. e TOMOV, S. ***clMAGMA: High Performance Dense Linear Algebra with OpenCL***. Technical Report UT-CS-13-706, University of Tennessee, March 2013.

CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H. e SKADRON, K. ***Rodinia: A benchmark suite for heterogeneous computing***. Em *IISWC*, págs. 44–54, 2009.

COLELLA, P. ***defining software requirements for scientific computing***. DARPA HPCS presentation, 2004.

COMCIDIS. ***ComCiDis, Computação Científica Distribuída***. `http://comcidis.lncc.br/`, 2013. [Online; accessado em 19-Janeiro-2013].

COULOURIS, G., D. J. e KINDBERG, T. ***Distributed Systems: Concepts and Design.*** Em *Addison-Wesley, 1994.*, 1994.

CREASY., R. J. ***The Origin of the VM/370 Time-sharing System.*** Em *IBM Journal of Research and Development*, págs. 483–490, 1981.

DANTAS, M. ***computação distribuída de alto desempenho: redes, grids e clusters computacionas***. Axcel Books, 2005.

DOLBEAU, R., BODIN, F. e DE VERDIERE, G. ***one opencl to rule them all?*** Em *Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on*, págs. 1–6, Sept 2013.

DONGARRA, J., MEUER, H. e STROHMAIER, E. ***Top500 Supercomputer Sites (13th edition)***. Technical Report UT-CS-99-425, Jun 1999. URL `http://www.top500.org`.

DUNCAN, R. ***A survey of parallel computer architectures.*** Em *IEEE Computer*, págs. 5–16, 1990.

EIJKHOUT, V. ***Introduction to High Performance Scientific Computing***. `http://tacc-web.austin.utexas.edu/veijkhout/public-html/istc/istc.html`, 2013. [Online; accessado em 19-Agosto-2013].

ENGEN, V., PAPAY, J., PHILLIPS, S. C. e BONIFACE, M. ***Predicting application performance for multi-vendor clouds using dwarf benchmarks***. Em *Proceedings of the 13th international conference on Web Information Systems Engineering*, WISE'12, págs. 659–665, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-35062-7.

FLYNN, M. ***Some Computer Organizations and Their Effectiveness.*** Em *IEEE Trans. Computer*, pág. 948, 1972.

FRIGO, M., STEVEN e JOHNSON, G. ***The design and implementation of FFTW3***. Em *Proceedings of the IEEE*, págs. 216–231, 2005.

HEROUX, M. A. e DONGARRA, J. ***toward a new metric for ranking high performance computing systems***. *SAND2013 - 4744*, jun 2013.

HEY, T., S. e TOLLE., K. ***The Fourth Paradigm: data-intensive scientific discovery.*** Redmond, Washington: Microsoft Research., 2009.

HOOPOE, H. ***HOOPOE CLOUD***. `http://www.cass-hpc.com/solutions/hoopoe`, 2013. [Online; accessado em 18-Janeiro-2013].

INTEL. ***Intel Invokes Phi to Reach Exascale Computing by 2018***. `http://www.pcworld.com/article/257792/`, 2011. [Online; accessado em 11-Janeiro-2013].

KAISER, A., WILLIAMS, S., MADDURI, K., IBRAHIM, K., BAILEY, D., DEMMEL, J. e STROHMAIER, E. ***TORCH Computational Reference Kernels: A Testbed for Computer Science Research***. Technical Report UCB/EECS-2010-144, EECS Department, University of California, Berkeley, Dec 2010. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-144.html`.

KIRK, B., H. W. ***Programando para processadores paralelos.*** Em *Ed. Campus*, 2011.

KVM. ***KVM, Kernel Based Virtual Machine***. `http://www.linux-kvm.org/`, 2011. [Online; accessado em 11-Janeiro-2013].

MACKAY, D. ***Optimization and Performance Tuning for Intel Xeon Phi Coprocessors - Part 1: Optimization Essentials***. `http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization`, 2013. [Online; accessado em 19-Janeiro-2013].

MATSUNAGA, A. e FORTES, J. A. B. ***On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications***. Em *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, págs. 495–504, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4039-9. URL `http://dx.doi.org/10.1109/CCGRID.2010.98`.

MIC. ***(MIC) - Many Integrated Core Architecture - Advanced***. `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html`, 2013. [Online; accessado em 11-Janeiro-2013].

MORIMOTO, C. ***Coprocessador aritmético.*** `http://www.hardware.com.br/livros/hardware-manual/coprocessador-aritmetico.html`, 2013. [Online; acces-

sado em 19-Janeiro-2013].

MURALEEDHARAN, V. ***Hawk-i HPC Cloud Benchmark Tool***. Msc in high performance computing, University of Edinburgh, Edinburgh, August 2012.

NIMBIX, N. ***NIMBIX Accelerated Compute Cloud - HPC Workloads in the Cloud***. `www.nimbix.net`, 2013. [Online; accessado em 19-Agosto-2013].

PHI. ***Intel Xeon Phi coprocessors accelerate the pace of discovery and innovation***. `http://blogs.intel.com/technology/2012/06/intel-xeon-phi-coprocessors-accelerate-discovery-and-innovation/`, 2012. [Online; accessado em 11-Janeiro-2013].

PHILLIPS, S. C., ENGEN, V. e PAPAY, J. ***Snow White Clouds and the Seven Dwarfs***. Em *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, págs. 738–745, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4622-3.

QUMRANET. ***Qumranet joins Red Hat***. `http://www.redhat.com/promo/qumranet/`, 2011. [Online; accessado em 11-Janeiro-2013].

S. CADAMBI, G. COVIELLO, C.-H. L. R. P. K. R. M. S. e CHAKRADHAR, S. ***cosmic: middleware for high performance and reliable multiprocessing on xeon phi coprocessors***. Em *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing, ser. HPDC 13*, págs. 215 – 226, 2013.

SAX, J. ***Parallel Dwarfs***. `http://paralleldwarfs.codeplex.com/`, 2013. [Online; accessado em 10-Dezembro-2013].

SHALF, J. ***SciDAC Review 14 - Dwarfs.*** `http://www.scidacreview.org/0904/html/ multicore1.html`, 2009. [Online; accessado em 22-Janeiro-2013].

SPRINGER, P. ***Berkeley's Dwarfs on CUDA***. Technical report, RWTH Aachen University, 2011. Seminar Project.

STRATTON, J. A., RODRIGRUES, C., SUNG, I.-J., OBEID, N., CHANG, L., LIU, G. e HWU, W.-M. W. ***parboil: a revised benchmark suite for scientific and commercial throughput computing***. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, março 2012.

TOP500. ***TOP 500 Supercomputers Sites***. `http://www.top500.org`, 2013. [Online; accessado em 10-Dezembro-2012].

VUDUC, R. ***OSKI: Optimized Sparse Kernel Interface***. `http://bebop.cs.berkeley.edu/oski/about.html`, 2006. [Online; accessado em 11-Janeiro-2013].

XEN. ***The home of XEN hypervisor***. `http://www.xen.org/`, 2011. [Online; accessado em 11-Janeiro-2013].