Avaliação de arquiteturas *manycore* e do uso da virtualização de GPUs

Laion L. F. Manfroi[1,2], Bruno Schulze[2], Raquel C. G. Pinto[1], Antonio R. Mury[2], Mariza Ferro [2]

[1]Sistemas e Computação - Instituto Militar de Engenharia
Praça Gen. Tibúrcio, 80 Urca, Rio de Janeiro - RJ, 22290-270 - Brasil

[2]Coordenação de Ciência da Computação - Laboratório Nacional de Computação Científica
Av. Getúlio Vargas, 333, Quitandinha, Petrópolis, RJ - Brasil

{laion, schulze, aroberto, mariza}@lncc.br, raquel@ime.eb.br

Resumo. *Nowadays virtualization is present from the various strategies for resource consolidation in data centers, to 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 on 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, a comparison between different architectures available in current HPC market is established.*

Resumo. *Atualmente a virtualização encontra-se presente desde nas diversas estratégias para a consolidação de recursos em Datacenters, até 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 resultados tratando-se em como a camada de virtualização afeta o acesso a estes dispositivos. 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 disponívies no mercado atual de HPC.*

1. Introdução

A exigência de um maior desempenho das aplicações faz com que a Computação de Alto Desempenho (HPC - *High Performance Computing*) esteja sempre em constante evolução. No passado, a base da HPC eram processadores não dedicados de uso genérico, que executavam todos tipos de instruções. Atualmente, os processadores gráficos (GPUs - *Graphics Processing Units*) são os responsáveis pelo alto desempenho de grande parte dos supercomputadores mais poderosos [Dongarra et al. 1999].

As GPUs modernas são capazes de alcançar taxas de processamento de diversas ordens de magnitude maiores que as CPUs (*Central Processing Units*) de propósito geral. Alcançar esta 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 aceleradores com muitos núcleos (*manycore*).

Uma maneira prática de gerenciar ambientes de alto desempenho destinados a multi-usuários e que se baseiam nestas arquiteturas de processamento paralelo envolve o uso da virtualização e o conceito de computação em nuvem. Esta infraestrutura provê benefícios, tais como abstração de recursos, infraestrutura elástica e orientada a serviço, facilidade no gerenciamento de recursos e dinamismo na disponibilização de um 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 o 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*), que possibilita dedicar um dispositivo de entrada e saída (E/S), exclusivamente a uma MV, com acesso direto e exclusivo ao dispositivo. Sendo assim, pode-se 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.

O objetivo deste trabalho é a pesquisa inicial e o desenvolvimento de testes na implantação de um ambiente para execução de GPUs virtualizadas, fazendo uso do a-tual estado da arte da tecnologia de IOMMU. A princípio, este ambiente será baseado em 2 soluções de virtualização que oferecem a capacidade de acesso direto ao dispositivo de E/S (IOMMU). Também buscamos como contribuição, o comparativo entre as principais arquiteturas paralelas (aceleradores) e 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 nas principais arquiteturas disponíveis atualmente no mercado de HPC.

Por meio de testes nestes ambientes, será realizado um comparativo inicial de execuções do uso de GPUs em máquinas reais e em máquinas virtuais, através de um algoritmo que pode ser classificado como *Dwarf*, um novo conceito utilizado na análise de desempenho. Foi escolhida a classe de *Dwarfs* DLA (*Dense Linear Algebra*), por ser a classe capaz de englobar o maior número de aplicações científicas [Asanovic et al. 2006], sendo empregada como o ponto inicial dos experimentos com os *Dwarfs*.

A investigação como cada solução de virtualização trata o acesso direto a determinado dispositivo de E/S através do IOMMU será feita como base para definição do tipo de arquitetura (baseada em GPU ou *big-cores* x86) e qual a solução de virtualização é a mais apropriada para cada tipo de aplicação definida pelos *Dwarfs*.

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

2. Trabalhos relacionados

O consumo de *hardware* (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 executar o maior número de aplicações sem a contenção de recursos, podendo auxiliar na estimativa do tempo de espera em sistemas de execução em fila, e, até mesmo, prover uma estimativa do custo da execução de uma aplicação em um ambiente de nuvem [Matsunaga and Fortes 2010]. Além disso, e o mais importante do ponto de vista deste trabalho, pode identificar a melhor arquitetura e o melhor ambiente para executar uma aplicação.

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 7 *Dwarfs* da computação científica. Os *Dwarfs* foram definidos para caracterizar comportamentos entre diferentes aplicações de HPC. 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 esta classificação para 13, após examinarem importantes domínios de aplicações, com interesses na aplicação dos *Dwarfs* para um maior número de problemas computacionais [Asanovic et al. 2006].

Alguns trabalhos possuem semelhanças com o trabalho proposto, 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 conjuntos de *Dwarfs* para serem usados em *benchmarks* ou *kernels*. Em [Asanovic et al. 2006] isto é feito para o SPEC2006 e EEMBC. Em [Che et al. 2009] isto é feito para alguns *kernels* especializados em CPU e GPU e em [Stratton et al. 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 no nível da camada de virtualização (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 et al. 2011] e [Engen et al. 2012], onde os *Dwarfs* são usados para prever o desempenho somente em ambientes de nuvem.

Enquanto alguns dos trabalhos citados focam exclusivamente em arquiteturas *manycore*, *multi-core* ou em ambientes virtualizados, este trabalho propõe como contribuição um maior e mais completo ambiente de experimentos, onde são 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.

É necessário conhecer o modelo de aplicações para escolher um *benchmark* (ou um conjunto deles) mais apropriado. Na próxima Seção são apresentados os objetivos e a descrição das avaliações utilizadas neste trabalho.

3. Contribuições e 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 [Che et al. 2009]. 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. Os objetivos destes experimentos são:

- Explorar o comportamento dos aceleradores (baseados em GPU e em x86);
- Comparar o comportamento dos aceleradores com o comportamento de uma CPU disposta de um número de núcleos semelhantes ao número de núcleos do acelerador x86;
- Explorar o efeito que a camada de virtualização (KVM e XEN) exerce frente à execução de aplicações científicas utilizando a técnica de acesso direto à GPU;
- Identificar quais os aspectos das abordagens de implementação (CUDA e OpenCL) que influenciam na execução e no tempo de resposta de um ambiente virtual;

A implantação deste ambiente e a análise foram executados utilizando a infraestrutura do grupo de Computação Científica Distribuída [ComCiDis 2013], situado no Laboratório Nacional de Computação Científica (LNCC). Na Subseção seguinte são descritas as especificações de cada arquitetura utilizada neste trabalho.

3.1. Arquiteturas utilizadas

- Arquitetura *multi-core* CPU: A arquitetura de CPU que compõe o ambiente *multi-core* utilizado neste trabalho é um sistema com 64 núcleos reais, divididos em 4 processadores, 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 instruções. Cada núcleo tem sua própria L1 de 16KB. A escolha desta arquitetura se baseou no seu número de núcleos, que se equipara à quantidade de núcleos reais do acelerador x86.

- Acelerador *manycore* baseado em x86: O acelerador baseado em x86 utilizado neste trabalho é disposto de 60 núcleos reais. Cada núcleo suporta 4 HW *threads*. A arquitetura é formada por núcleos x86 com unidades de vetorização de 512 bits, executando-os a aproximadamente 1GHz, chegando ao pico de desempenho de 1 TeraFlop (TF) com precisão dupla, além de possuir um sistema operacional (SO) Linux customizado. Este acelerador 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 este acelerador possui seu próprio SO, ele é capaz de executar as aplicações nativamente, trabalhando como um nó de execução independente através da compilação do código no seu hospedeiro e da transferência dos arquivos executáveis para o acelerador.

- Acelerador baseado em GPU: O acelerador baseado em GPU usado neste trabalho é dedicado exclusivamente para o processamento de aplicações científicas, 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/sec 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.

3.2. Descrição dos experimentos

Nos experimentos contidos neste trabalho foram utilizadas as implementações contidas na *suite* Rodinia e nenhuma configuração especial foi feita para executar nas CPUs e nos aceleradores. As alterações pontuais 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 ficaram superiores a 99%, 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 as matrizes estivessem disponíveis na memória de todos os aceleradores utilizados nos testes. As matrizes quadradas utilizadas de entrada foram definidas com os seguintes tamanhos: 1024x1024, 2048x2048, 4096x4096, 8192x8192 e 16384x16384. Na Subseção seguinte são mostradas as especificações de cada experimento:

3.2.1. Experimentos no ambiente real

No ambiente real, o principal objetivo é a medição do desempenho atingido quando são utilizadas arquiteturas no padrão x86 (CPU *multi-core* e acelerador *manycore*) 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 et al. 2013] e [Dolbeau et al. 2013]. Nestes testes foram utilizadas as versões padrões das implementações do LUD em OpenMP e OpenCL, contidas no Rodinia.

Para que seja possível definir uma comparação entre as arquiteturas x86 (CPU e acelerador) e verificar a escalabilidade atingida, foi definida uma porcentagem de número de núcleos utilizados (25%, 50%, 75% e 100%). Na prática, isto significa que um sistema com 64 núcleos disponíveis irá executar com 50% se 32 núcleos estiverem sendo utilizados pela aplicação, da mesma maneira, irá operar com 25% se 16 núcleos estiverem sendo utilizados.

3.2.2. Experimentos no ambiente virtual

No ambiente virtual, o primeiro experimento foi feito com a intenção de avaliar o comportamento e o impacto do 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 nestes testes nos oferecem uma base de referência para investigar o atual estágio das tecnologias de IOMMU e *PCI passthrough* (implementação do IOMMU a nível de *software*), com a possibilidade de verificar o comportamento dos resultados alcançados com cada linguagem de programação. A análise busca procurar diferenças em como o hipervisor trata o acesso ao dispositivo quando executado algum problema relacionado a um *dwarf* DLA, implementado em alguma linguagem específica.

Em ambos os 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. Além disso, as MVs foram virtualizadas com os hipervisores KVM e XEN, executando no mesmo hospederio (sem concorrência), com a mesma configuração (10GB de memória RAM, 10 núcleos e 50GB disponíveis para sistemas de arquivos), utilizando o mesmo processador do servidor hospedeiro. Na próxima Seção são apresentados os resultados e a análise realizada com base nos experimentos aqui descritos.

4. Análise dos resultados

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

4.1. Resultados no ambiente real

As figuras 1, 2a e 2b mostram a relação entre o tempo de execução (em segundos) e a porcentagem de uso do número de núcleos. Neste primeiro experimento somente são mostrados os gráficos para os tamanhos 1024x1024, 4096x4096 e 16384x16384 de matrizes de entrada, pois são os casos onde são encontradas as mudanças mais significativas.

No primeiro experimento no ambiente real, pode-se notar que utilizando uma matriz de 1024x1024 (Figura 1) como entrada, o tempo de execução no acelerador x86 foi pior nos casos de uso de uma quantidade grande de núcleos (50%, 75% e 100%). Para 25% de alocação dos núcleos, a CPU *multi-core* x86 gasta 20% mais tempo do que o acelerador, sendo o único caso em que a CPU ficou com pior tempo de execução (para este tamanho de matriz). Estes comportamentos se devem à maior demora na alocação dos

núcleos dentro do acelerador, o que gera pouco impacto no tempo de execução total (*Wall time*) com poucos núcleos sendo utilizados. Contudo, a partir do aumento da utilização dos núcleos disponívieis (50% ou mais), este tempo de alocação para o acelerador x86 gera um maior impacto no tempo 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 neste acelerador, o que resultou em pouco tempo dedicado exclusivamente ao processamento. Isto é, a partir do momento em que todos os núcleos do acelerador x86 estavam disponíveis para processar e com os dados alocados, o problema finalizava a sua execução devido ao seu pequeno tamanho.

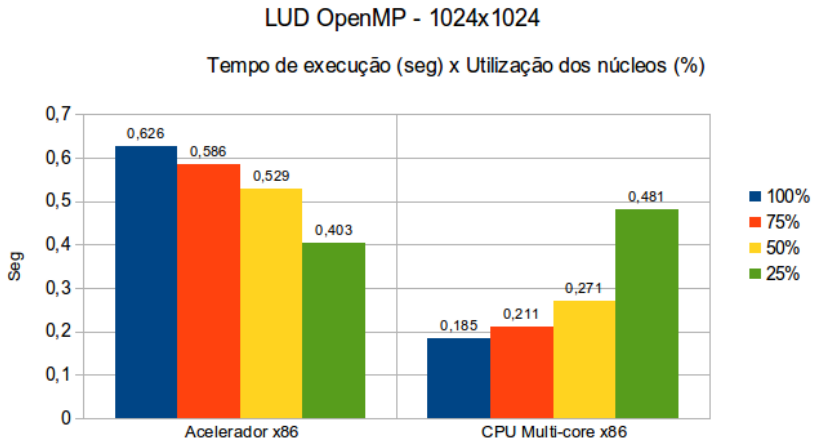

Figura 1. Teste de desempenho com matriz 1024x1024 em OpenMP.

Quando o tamanho da matriz é aumentado para 4096x4096 e 16384x16384, pode ser notado 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 4096x4096 (Figura 2a) e chegando a utilizar até 65% menos tempo para as matrizes 16384x16384 (Figura 2b), como melhores casos. Na figura 2a 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, tendo como principal agravante ainda o tempo de alocação dos núcleos com a pequena matriz de entrada. A escalabilidade melhora quando o problema tem o seu tamanho de problema de entrada aumentado, no caso da matriz 16384x16384 (figura 2b).

O segundo experimento submetido a este ambiente busca a relação entre o tempo de execução e os tamanhos das matrizes de entrada para a CPU *multi-core* x86, o acelerador x86 e o acelerador GPU.

A mesma implementação em OpenCL foi submetida para todas arquiteturas. Na tabela 1 é possível verificar que o acelerador baseado em GPU obteve sempre os melhores resultados. Isto se deve ao grande 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 o caso da matriz 1024x1024, onde as arquiteturas baseadas em x86 (CPU e acelerador) obtiveram tempos de execução

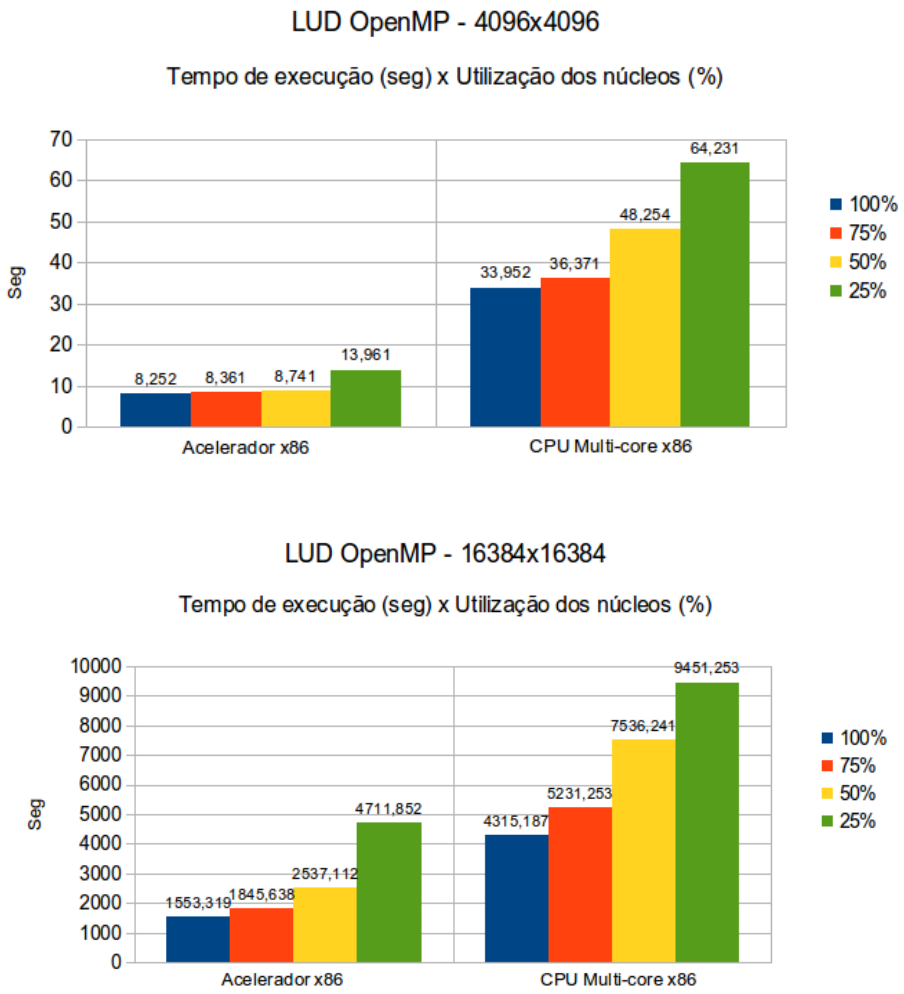

Figura 2. Teste de desempenho com matriz 4096x4096 (2a) e 16384x16384 (2b) em OpenMP.

semelhantes, enquanto o acelerador baseado em GPU foi cerca de 4 vezes mais rápido. Com o aumento do tamanho das matrizes de entrada, o acelerador baseado em GPU obteve sempre os melhores resultados, um exemplo é o caso da matriz 16384x16384, onde executou cerca de 7 vezes mais rápido que a CPU *multi-core* e 3,2 vezes mais rápido que o acelerador x86. 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 foi sempre o melhor entre todas as arquiteturas analisadas neste trabalho.

4.2. 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 3 e 4). Isto se deve à maturidade das tecnologias de *PCI passthrough* (usada em ambos os hipervisores) e de IOMMU implementado no *hardware* utilizado.

Tabela 1. Tempos médios de execução em OpenCL para todas as 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

Os níveis semelhantes de execução são alcançados principalmente pelo emulador QEMU, utilizado em ambos hipervisores e personalizado de acordo com as características próprias de cada um, refletindo diretamente na maneira com que cada SO convidado trata o acesso aos dispositivos de E/S.

No primeiro experimento para o ambiente virtual, foi utilizada a implementação em CUDA do algoritmo LUD, que foi executado no sistema hospedeiro nativamente (sem a camada de virtualização), em uma MV criada com o KVM e em uma MV 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 3), verifica-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 demonstrou-se com o uso das matrizes 1024x1024, onde o uso da camada de virtualização acarretou uma perda de 5% comparado à execução nativa no hospedeiro. Além disso, notamos que o melhor caso de execução foi com o uso das matrizes 16384x16384, onde a perda foi 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 virtualização em ambientes que possam utilizar periféricos de E/S dedicados exclusivamente a uma MV. Uma das grandes contribuições para estes resultados é a proximidade do KVM com o *kernel* do SO. 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).

Comparando os tempos obtidos com o XEN à execução no hospedeiro sem a ca-

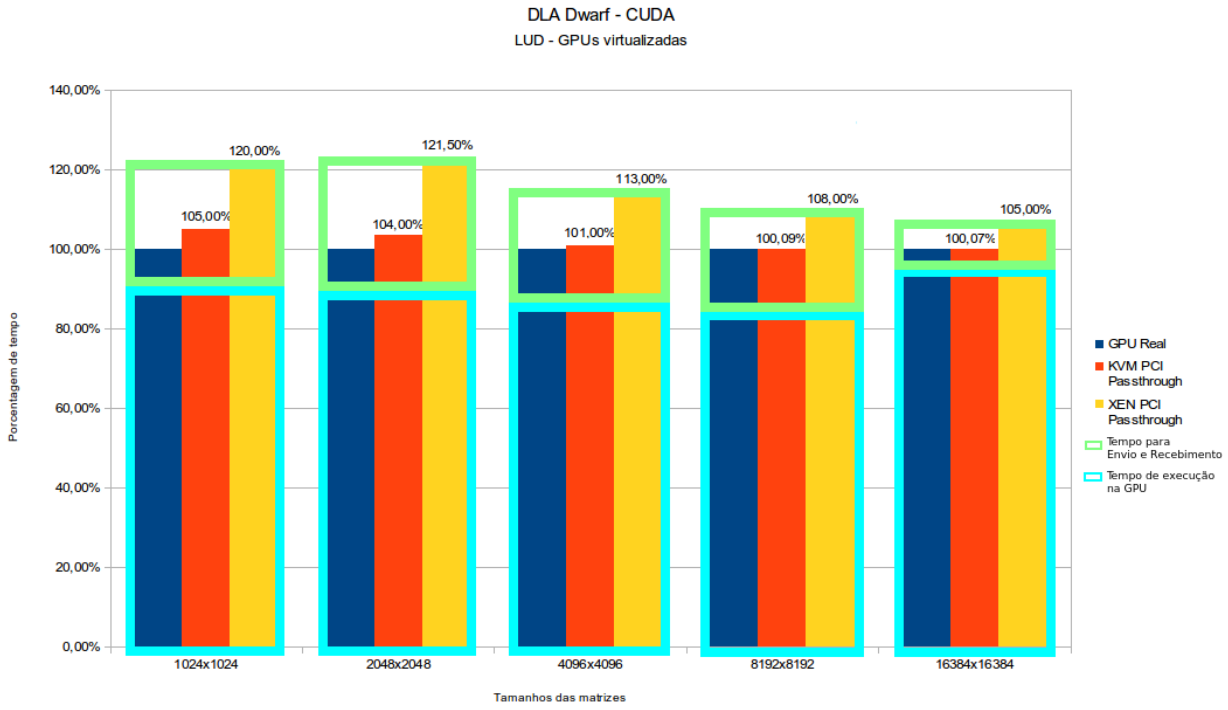

Figura 3. Porcentagem de tempo para envio/recebimento e execução.

mada de virtualização, nota-se que as taxas de perda para este hipervisor foram sempre maiores, chegando aos 21,5% para as matrizes de 2048x2048. Conforme o tamanho dos problemas de entrada aumentam, esta taxa diminui, sendo possível verificar que no caso para matrizes 16384x16384 a perda com o uso do XEN se estabeleceu em 5% comparado ao hospededeiro.

Comparando os tempos obtidos entre os hipervisores, temos que o XEN apresentou menor desempenho, com o pior caso para as matrizes 2048x2048, 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 MV, porém acarreta em uma sobrecarga nos tempos de envio/execução/recebimento das informações para o dispositivo.

Na Figura 3, 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 MV 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, nota-se que o hipervisor XEN obteve os maiores tempos necessários para isto, permanecendo muito acima aos outros ambientes, chegando a utilizar 32% do tempo total para enviar e receber os dados a serem processados, enquanto os outros ambientes não passaram de 15% do tempo utilizados para esta finalidade. Nesta Figura são mostradas as porcentagens dos tempos levando em

consideração as porcentagens totais obtidas na Figura 3.

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 (*Live Migration*) das máquinas virtuais, porém acarreta em perda de desempenho no uso do *PCI Passthrough*.

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 4, mais uma vez o ambiente real obteve os melhores resultados dentre todos os ambientes.

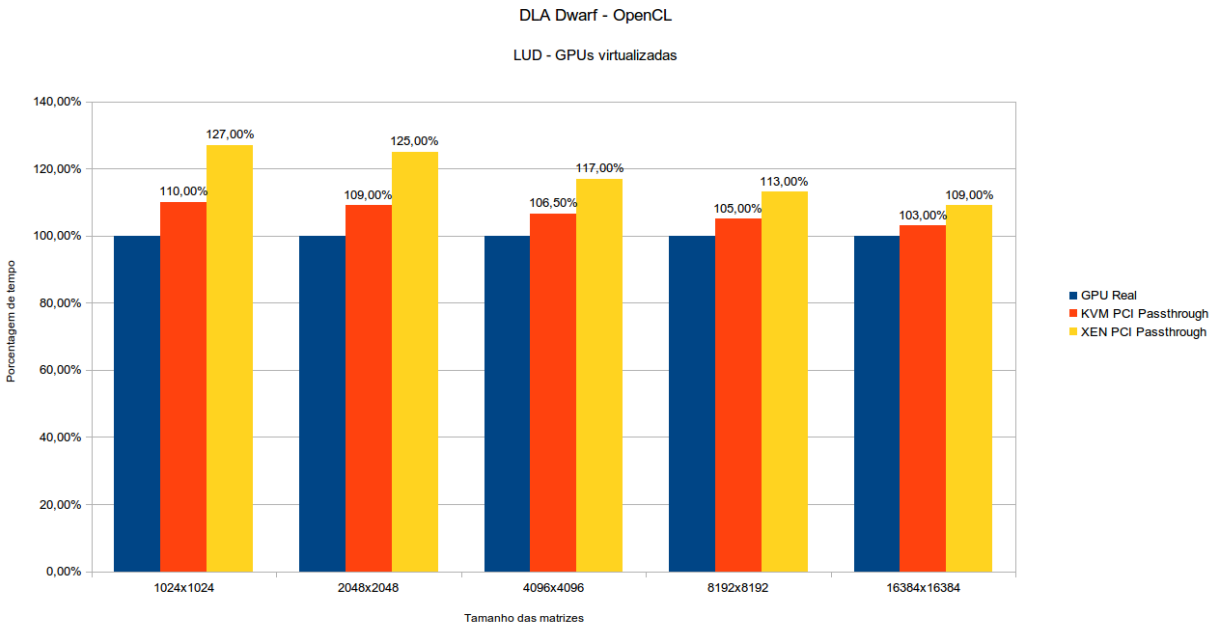

Figura 4. Comparativo de desempenho em OpenCL nos ambientes virtuais X ambiente real.

Comparando os tempos obtidos com o XEN à execução no Hospedeiro sem a camada de virtualização, notamos que as taxas de perda para ambos os hipervisores foram maiores das obtidas anteriormente com a implementação em CUDA. Pode-se notar que o uso do KVM acarretou em perdas de até 10% (para 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% (1024x1024 e 2048x2048) 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 (descritas a seguir). Estas características acarretaram em taxas maiores de perda nos ambientes virtuais, se comparados ao experimento realizado em CUDA.

Na implementação em CUDA, 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 nas implementações em OpenCL.

As grandes diferenças entre as implementações e onde se encontram as maiores taxas de perda dos resultados em OpenCL são resultado da necessidade da chamada de funções responsáveis pela definição dos argumentos e do tamanho dos blocos que cada *kernel* irá processar.

Como o desenvolvimento do OpenCL teve, como principal requisito, o grande número de *devices* habilitados a executar um código em OpenCL, é necessário um controle mais complexo dos ambientes 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 hospedeiro controla sua execução através de um objeto chamado *command-queue*. 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'. Através disto, notou-se que estas funções de definição de argumentos e controle da *command-queue* e do *context* geram um controle excedente feito pela CPU. Como as instruções são enviadas a partir de um ambiente virtual, o tratamento dos hipervisores geram o *overhead* encontrado nos casos utilizando a versão em OpenCL do problema LUD.

Pode ser verificado que uma implementação em OpenCL é muito mais dependente da CPU, se comparada à 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 4, 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.

5. Considerações finais

A proposta apresentada neste trabalho teve como objetivo uma análise inicial comparativa entre os principais aceleradores disponíveis no atual mercado de HPC. Além disto, o objetivo também foi 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 também demonstrou as taxas de perda/ganho que podem ser atingidas, além de definir qual o tipo de camada de virtualização melhor se comporta quando opera com acessos diretos aos aceleradores, que são parte fundamental das arquiteturas atuais de HPC.

5.1. Conclusões e trabalhos futuros

O estudo apresentado neste trabalho busca uma análise comparativa entre os ambientes real e virtual, onde a intenção é comparar o que temos como atual estado da arte de aceleradores e hipervisores. Isto foi feito utilizando a *suite* Rodinia, que é baseada na classificação dos *Dwarfs* 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.

A partir dos resultados alcançados com estes experimentos no ambiente real, pode-se concluir que a GPU obteve o 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 4 vezes mais rápida para o primeiro tamanho de matriz neste experimento. Com o aumento do tamanho das matrizes de entrada, a GPU obteve sempre os melhores resultados, um exemplo é o caso da matriz 16384x16384, onde executou cerca de 7 vezes mais rápido que a CPU *multi-core* e 3,2 vezes mais rápido que o acelerador x86. Com isso, temos que 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, temos o esforço do aprendizado das estratégias de programação em GPUs (CUDA ou OpenCL), podendo acarretar em um tempo maior de implementação do projeto.

Também foram submetidos dois experimentos ao ambiente virtual, ambos buscaram avaliar como os hipervisores tratam o acesso direto à dispositivos de E/S. Com estes experimentos 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.

Como trabalhos futuros, pretende-se avaliar o comportamento das mesmas arquiteturas quando executam outros tipos de classes de *Dwarfs*. Por conseguinte, será avaliado o comportamento de aplicações específicas de grupos de pesquisas parceiros ao ComCiDis, que podem ser caracterizadas como o mesmo *Dwarf*. A principal motivação para isto é buscar 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.

Referências

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

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., and Yelick, K. A. (2006). The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Cao, C., Dongarra, J., Du, P., Gates, M., Luszczek, P., and Tomov, S. (2013). clMAGMA: High Performance Dense Linear Algebra with OpenCL. Technical Report UT-CS-13-706, University of Tennessee.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54.

Colella, P. (2004). Defining software requirements for scientific computing. DARPA HPCS presentation.

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

Dolbeau, R., Bodin, F., and de Verdiere, G. (2013). *One OpenCL to rule them all?* In *Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on*, pages 1–6.

Dongarra, J., Meuer, H., and Strohmaier, E. (1999). Top500 Supercomputer Sites (13th edition). Technical Report UT-CS-99-425.

Engen, V., Papay, J., Phillips, S. C., and Boniface, M. (2012). Predicting application performance for multi-vendor clouds using dwarf benchmarks. In *Proceedings of the 13th international conference on Web Information Systems Engineering*, WISE'12, pages 659–665, Berlin, Heidelberg. Springer-Verlag.

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

Matsunaga, A. and Fortes, J. A. B. (2010). On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 495–504, Washington, DC, USA. IEEE Computer Society.

Muraleedharan, V. (2012). Hawk-i HPC Cloud Benchmark Tool. Msc in high performance computing, University of Edinburgh, Edinburgh.

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

Phillips, S. C., Engen, V., and Papay, J. (2011). Snow White Clouds and the Seven Dwarfs. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 738–745, Washington, DC, USA. IEEE Computer Society.

Springer, P. (2011). Berkeley's Dwarfs on CUDA. Technical report, RWTH Aachen University. Seminar Project.

Stratton, J. A., Rodrigrues, C., Sung, I.-J., Obeid, N., Chang, L., Liu, G., and Hwu, W.-M. W. (2012). Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana.