DOI: http://dx.doi.org/10.29069/forscience.2014v2n1.e53

CUDA VS. OPENCL: UMA COMPARAÇÃO TEÓRICA E TECNOLÓGICA

Lauro Cássio Martins de Paula[1]

RESUMO

Apresenta-se neste trabalho uma comparação entre duas arquiteturas para computação paralela: *Compute Unified Device Architecture* (CUDA) e *Open Computing Language* (OpenCL). Alguns trabalhos na literatura apresentaram uma comparação de desempenho computacional entre as duas arquiteturas. Entretanto, ainda não existe nenhum artigo recente e completo que destaca claramente qual arquitetura pode ser considerada a mais eficiente. O objetivo deste trabalho é realizar uma comparação apenas em nível de *hardware*, *software*, tendências tecnológicas e facilidades de utilização, evidenciando aquela que pode apresentar o melhor desempenho de uma maneira geral. Para tal, descreve-se os principais trabalhos que já fizeram uso de pelo menos uma das arquiteturas. Observou-se que, por ser um sistema heterogêneo, a escolha do OpenCL pode parecer mais óbvia. No entanto, foi possível concluir que CUDA, apesar de poder ser utilizada apenas nas placas gráficas da NVIDIA®, tem sido uma referência e mais utilizada ultimamente.

Palavras-chave: CUDA. OpenCL. GPU.

CUDA VS. OPENCL: A THEORETICAL AND TECHNOLOGICAL COMPARISON

ABSTRACT

This paper presents a comparison between two architectures for parallel computing: Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL). Some works in the literature have presented a computational performance comparison of the two architectures. However, there is not some complete and recent paper that highlights clearly which architecture can be considered the most efficient. The goal is to make a comparison only in level of hardware, software, technological trends and ease of use, highlighting one that may present the best cost-effective in general. To this end, we describe the main works that have used at least one of the architectures. It was observed that the choice of OpenCL may seem more obvious for being a heterogeneous system. Nevertheless, it was concluded that CUDA, although it can be used only in graphics cards from NVIDIA®, has been a reference and more used recently.

Keywords: CUDA. OpenCL. GPU.

[1] Mestre em Ciência da Computação pela Universidade Federal de Goiás (UFG) e bacharel em Ciência da Computação, com ênfase em Matemática Computacional pela Pontifícia Universidade Católica de Goiás (PUC Goiás). E-mail: laurocassio21@gmail.com.

1 INTRODUÇÃO

Ao longo dos anos, tem-se acompanhado uma forte evolução tecnológica e computacional. A Computação Paralela (CP) tem contribuído bastante para essa evolução e também para diversas áreas da ciência, que vão desde simulações computacionais a aplicações científicas como, por exemplo, aplicativos para mineração de dados e processamento de transações. A CP é uma forma eficiente do processamento da informação, com ênfase na exploração de eventos concorrentes no processo computacional.

Com o avanço da tecnologia, novas arquiteturas computacionais têm sido desenvolvidas. Soluções com vários processadores em uma mesma placa vêm sendo elaboradas, e processadores com vários núcleos de processamento são a nova tendência tecnológica na atualidade (SMITH, 2011). Existem muitas razões para a utilização da CP, destacando-se três delas:

a) execução de várias atividades simultaneamente;
b) capacidade de resolução de problemas maiores;
c) aumento do desempenho computacional.

Tradicionalmente, a computação paralela foi motivada pela resolução (ou simulação) de problemas com grande relevância científica e econômica, denominados *Grand Challenge Problems* (GCP). Tipicamente, os GCPs simulam alguns fenômenos que não podem ser medidos por experimentação (fenômenos climáticos, físicos, químicos, dentre outros). Nesses casos, as aplicações estão exigindo o desenvolvimento de processadores cada vez mais rápidos. Isso ocorre porque a maioria dessas aplicações requer um grande esforço computacional para processar grandes quantidades de dados.

Há fascinantes aplicações e excelentes oportunidades onde a CP pode apresentar ganhos de desempenho significativos. Fora do contexto científico, vivenciamos também uma explosão no volume de dados. Por exemplo, as corporações buscam cada vez mais ferramentas que transformem esses dados em informações valiosas, com o objetivo de fornecer respostas às necessidades de seus negócios e as ajudem a otimizar seus processos (KIRK, 2008). Nos últimos anos, a expectativa da ciência e do mercado indica a necessidade de redes de computadores cada vez mais rápidas, sistemas distribuídos altamente escaláveis e arquiteturas de computadores multiprocessados, mostrando claramente que o processamento paralelo é o futuro da computação.

Atualmente, têm-se desenvolvido dois tipos de processadores: *multicore* e *manycore*. Os processadores *multicore*, normalmente, contêm poucos núcleos, porém com um grande poder de processamento. Tais processadores visam minimizar a latência de memória, reservando uma parte do chip para memória cache, e permitem um uso moderado de linhas de execução (*threads*) (STALLINGS, 2002). Os processadores *manycore* são desenvolvidos com dezenas ou centenas de núcleos mais simples, otimizados para uma maior vazão na execução de instruções, utilizando centenas ou até mesmo milhares de *threads* (KIRK, 2008).

De forma correspondente à evolução do *hardware*, novos modelos de programação, capazes de aproveitar o poder desta nova tecnologia, têm sido elaborados, destacando-se dois deles: *Compute Unified Device Architecture* (CUDA) e *Open Computing Language* (OpenCL). Modelos de programação como CUDA e OpenCL permitem que aplicações possam ser executadas mais facilmente na GPU (PAULA et al., 2013b). Tanto em CUDA quanto em OpenCL, devido a uma ampla disponibilidade de *Application Programming Interfaces* (API), a implementação de aplicações paralelas eficientes pode ser facilitada, embora as GPUs ainda sejam mais difíceis de serem programadas que as CPUs (PAULA et al., 2013a). Isso ocorre porque, para a paralelização de dados em GPUs, a organização e o número de *threads* geralmente devem ser gerenciados manualmente pelo programador (PAULA, 2014a).

Muitos trabalhos na literatura têm utilizado CUDA e (ou) OpenCL para a solução de diversos tipos de problemas paralelizáveis. Após uma extensa pesquisa bibliográfica, foi possível observar que existem trabalhos que realizam apenas comparações de desempenho computacional entre ambos. Alguns mostram que CUDA supera o OpenCL na maioria das aplicações, e outros evidenciam que o OpenCL pode ser uma boa alternativa em relação a CUDA. Entretanto, uma comparação teórica e tecnológica entre esses dois *frameworks* ainda tem sido pouco realizada. Portanto, o objetivo deste trabalho é realizar uma comparação entre CUDA e OpenCL apenas em relação à aspectos de *hardware*, *software*, tendências tecnológicas e facilidades de utilização, evidenciando aquele que apresenta o melhor desempenho.

Este artigo está organizado da seguinte forma. A Seção 2 descreve os principais detalhes sobre uma unidade de processamento gráfico (GPU). Os principais aspectos sobre CUDA são abordados na Seção 3. A Seção 4 detalha a arquitetura do OpenCL. Uma comparação entre CUDA e OpenCL é apresentada na Seção 5. Por fim, a Seção 6 mostra as conclusões do trabalho.

2 UNIDADE DE PROCESSAMENTO GRÁFICO

Primordialmente, o computador foi desenvolvido como uma máquina sequencial. De forma análoga, a maioria das linguagens de programação requer que o programador especifique um algoritmo como uma sequência de instruções, que são executadas sequencialmente pelo processador. Cada instrução é executada como uma sequência de operações (STALLINGS, 2002), tais como:

a) busca e decodificação de instruções;
b) cálculo dos endereços dos operandos;
c) busca dos operandos na memória;
d) cálculo com os operandos;
e) armazenamento dos resultados na memória.

A Taxonomia (ou Classificação) de Flynn, definida por Michael J. Flynn em 1966, classifica a arquitetura computacional de acordo com o processamento do fluxo de instrução e de dados (FLYNN; RUDD, 1996). Como resultado, tem-se as quatro classes descritas a seguir:

a) *Single Instruction Single* Data (SISD): uma única unidade de controle é responsável por processar um único fluxo de instruções num único fluxo de dados;
b) *Single Instruction Multiple* Data (SIMD): projetos de arquitetura onde uma única instrução é executada simultaneamente em múltiplos fluxos de dados;
c) *Multiple Instruction Single Data* (MISD): este tipo de arquitetura é fonte de divergência entre pesquisadores da área de arquitetura de computadores. Navaux (1989) afirma que nenhum sistema conhecido se encaixa nesta categoria. Entretanto, Quinn (2003) aponta como exemplo para esta categoria o systolicarray;
d) *Multiple Instruction Multiple Data* (MIMD): arquiteturas que apresentam múltiplas unidades de processamento que manipulam diferentes fluxos de instrução com diferentes fluxos de dados.

As *Graphics Processing Units* (GPU) foram inicialmente desenvolvidas como uma tecnologia orientada à vazão, otimizada para cálculos de uso intensivo, onde muitas operações idênticas podem ser realizadas em paralelo sobre diferentes dados (SIMD) (NVIDIA CUDA,

2013). A computação com GPU pode ser considerada como sendo o uso de uma unidade de processamento gráfico como um coprocessador para acelerar as *Central Processing Units*(CPU) para computação científica e de propósito geral (PAULA, 2013d). Diferente de uma CPU *multicore*, a qual executa algumas *threads* em paralelo, a GPU foi projetada para executar milhares de *threads* (PAULA, 2013c). A Figura 1 mostra uma comparação entre a arquitetura de uma CPU com apenas quatro unidades lógica e aritmética (ULAs) e uma GPU com 128 ULAs.

Figura 1- Comparação entre a arquitetura de uma CPU e uma GPU
Fonte: NVIDIA CUDA (2013).

Devido à crescente utilização de alto poder computacional, a necessidade de processadores mais velozes vem se tornando inevitável. Desde 2003, as GPUs têm liderado a corrida do desempenho em ponto flutuante, como mostra o gráfico 1.

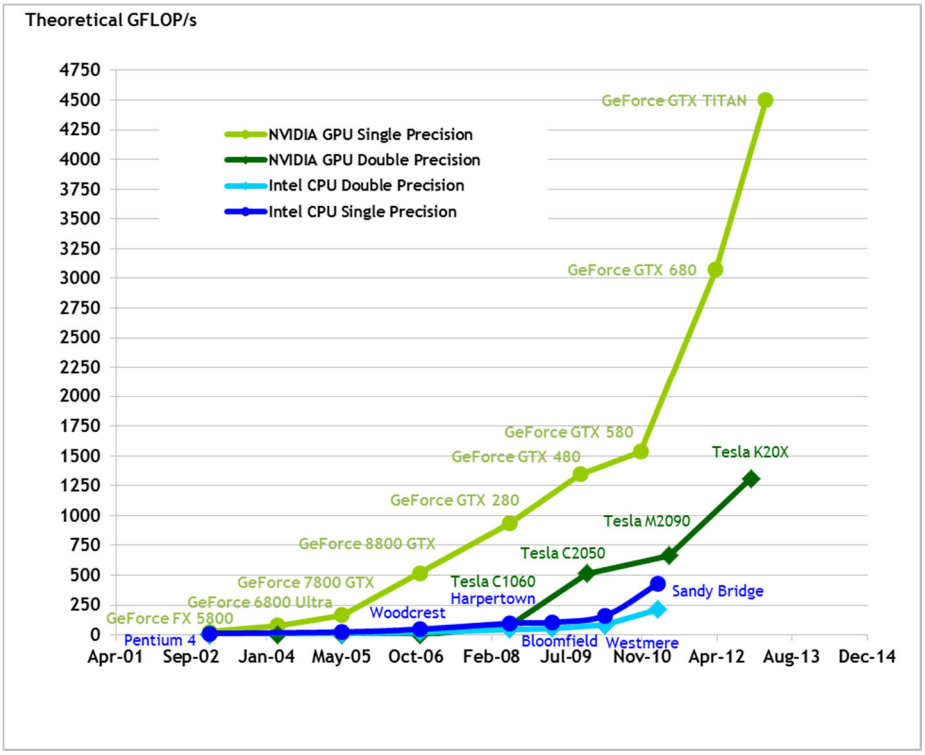

Gráfico 1 - Comparação de capacidades de processamento entre CPUs e GPUs
Fonte: NVIDIA CUDA (2013).

Por um lado, as GPUs são melhores adaptadas para endereçar problemas que podem ser expressos através de cálculos realizados de forma paralela (SMITH, 2011). Como o mesmo programa é executado para cada elemento de dado, há menos requisitos referentes a controles de fluxo e, exatamente por ser executado em muitos elementos de dados, a latência de acesso à memória pode ser ocultada pela realização de cálculos (GAIOSO et al., 2013). Além do desempenho, as GPUs contam com um importante fator para seu sucesso: a presença de mercado. Ter forte presença de mercado é fundamental para o sucesso de uma arquitetura paralela (KIRK, 2008).

Por outro lado, como toda tecnologia, as GPUs possuem suas limitações (PAULA et al., 2013a). Dependendo do volume de dados, o desempenho computacional da GPU pode se mostrar inferior quando comparado ao desempenho da CPU. Isso significa que a quantidade de dados a serem transferidos para a memória da GPU deve ser levada em consideração, devido à existência de um *overhead* associado à paralelização das tarefas na GPU (NVIDIA CUDA, 2009; KIRK, 2008; PAULA et al., 2013b). Fatores em relação ao tempo de acesso à memória também podem influenciar no desempenho computacional. Em outras palavras, o acesso à memória global da GPU, geralmente, apresenta uma alta latência e pode estar sujeito a um acesso aglutinado aos dados em memória (PAULA, 2013c).

A NVIDIA® e a AMD® são exemplos de empresas que desenvolvem GPUs e disputam o mercado de computação paralela. Como mostra as Seções 3 e 4, linguagens específicas para a GPU foram desenvolvidas por essas duas empresas.

3 ARQUITETURA PARA DISPOSITIVOS DE COMPUTAÇÃO UNIFICADA

CUDA foi a primeira API, criada pela NVIDIA® em 2006, a permitir que a GPU pudesse ser utilizada para uma ampla variedade de aplicações (NVIDIA CUDA, 2013). CUDA é suportada por todas as placas gráficas da NVIDIA®, que são extremamente paralelas, possuindo muitos núcleos com diversas memórias *cache* e uma memória compartilhada por todos os núcleos (KIRK, 2008). No ambiente de programação CUDA, o sistema computacional distingue entre o que é executado na CPU (*host*) e o que é executado na GPU (*device*). Um programa em CUDA consiste em partes executadas no *host* e outras partes executadas no *device*. A separação fica a cargo do compilador da NVIDIA® (*nvcc*) durante a compilação. O código em CUDA é uma extensão da linguagem computacional C

(CUDA-C), em que algumas palavras-chave são utilizadas para rotular as funções paralelas (*kernels*) e suas estruturas de dados (NVIDIA CUDA, 2009).

A implementação de uma função a ser executada em paralelo pelas *threads* nos núcleos da GPU é chamada *kernel*. Os *kernels* normalmente geram um grande número de *threads* para explorar o paralelismo de dados. O número de *threads* é especificado pelo programador na chamada da função. Quando um *kernel* é invocado, ele é executado como uma grade (*grid*) de *threads* paralelas (NVIDIA CUDA, 2013). Como ilustra a Figura 2, as *threads* em um *grid* são organizadas em uma hierarquia de dois níveis, onde cada *grid* consiste em um ou mais blocos de *threads*.

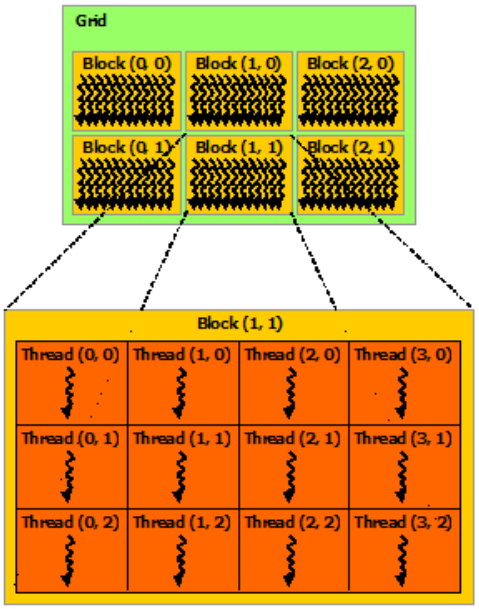

Figura 2 - *Grid* com vários blocos de *threads*
Fonte: NVIDIA CUDA (2013).

Por meio de uma classificação chamada *Compute Capability*, a NVIDIA® diferencia a capacidade de computação de cada GPU no ambiente CUDA. Por existir limitações da arquitetura da GPU, especificadas em cada revisão, tal classificação deve ser conhecida pelo programador (KIRK, 2008).

Desde o seu surgimento, diversos trabalhos têm utilizado CUDA para a paralelização de vários tipos de problemas. Por exemplo, Yldirim e Ozdogan (2011) apresentaram um algoritmo como uma abordagem de agrupamento baseado em transformada *wavelet* para paralelização em GPU usando CUDA-C. Fabris e Krohling (2012) propuseram um algoritmo de evolução implementado em CUDA-C para solução de problemas de otimização. Paula et al. (2013a) utilizaram CUDA-C para paralelizar o método BiCGStab(2), o qual é utilizado

para solução de sistemas lineares. Paula et al. (2013b) propuseram uma estratégia de paralelização para a fase 2 do Algoritmo das Projeções Sucessivas utilizando CUDA-C. Paula (2013c) utilizou a estratégia proposta em (PAULA et al., 2013a) e apresentou uma comparação entre métodos iterativos na solução de sistemas lineares grandes e esparsos. Gaioso et al. (2013), utilizando CUDA-C, apresentaram uma paralelização para o algoritmo Floyd-Warshall, utilizado para encontrar os caminhos mínimos entre todos os pares de vértices de um grafo. Por fim, Paula et al. (2014b) propuseram uma paralelização para um algoritmo (Firefly Algorithm) que seleciona variáveis na solução de problemas de calibração multivariada.

Recentemente, a MathWorks® desenvolveu um plugin capaz de fazer a integração entre CUDA e MATLAB. Fazer uso do MATLAB para computação em GPU pode permitir que aplicações sejam aceleradas mais facilmente (LITTLE; MOLER, 2013). As GPUs podem ser utilizadas com MATLAB por meio do *Parallel Computing Toolbox* (PCT). O PCT fornece uma maneira eficiente para acelerar códigos na linguagem MATLAB, executando-os em uma GPU. Para isso, o programador deve alterar o tipo de dado para entrada de uma função para utilizar os comandos (funções) do MATLAB que foram subcarregados (*GPUArray*). Por meio da função *GPUArray* é possível alocar dados na memória da GPU e fazer chamadas a várias funções do MATLAB, que são executadas nos núcleos de processamento da GPU. Além disso, os desenvolvedores podem fazer uso da interface *CUDAKernel* no PCT para integrar seus códigos em CUDA-C com o MATLAB (REESE; ZARANEK, 2011).

O desenvolvimento de aplicações a serem executadas na GPU utilizando o PCT é, geralmente, mais fácil e rápido do que utilizar a linguagem CUDA-C (LIU; CHENG; ZHOU, 2013). De acordo com Little e Moler (2013), isso ocorre porque vários aspectos de exploração de paralelismo são realizados pelo próprio PCT. Entretanto, a organização e o número de *threads* a serem executadas nos núcleos da GPU não podem ser gerenciados manualmente pelo programador. Ainda, é importante ressaltar que, para poder ser utilizado, o PCT requer uma placa gráfica da NVIDIA®.

Após a integração CUDA-MATLAB, alguns trabalhos têm utilizado essa tecnologia. Por exemplo, a NVIDIA® (2007) lançou um livro que demonstra como programas desenvolvidos em MATLAB podem ser acelerados usando suas GPUs. Simek e Asn (2008) apresentaram uma implementação em MATLAB com CUDA para compressão de imagens médicas. Kong et al. (2010) aceleraram algumas funções do MATLAB para processamento de

imagens em GPUs. Reese e Zaranek (2011) desenvolveram um manual de programação em GPUs usando MATLAB. Little e Moler (2013) mostraram vários detalhes de como realizar computações do MATLAB em GPUs usando CUDA. Mais recentemente, Liu, Cheng e Zhou (2013) apresentaram uma pesquisa e comparação de programação usando GPUs em MATLAB. Com base nesses resultados, nota-se que, futuramente, o PCT poderá ser mais utilizado devido ao fato de permitir que um código na linguagem MATLAB possa ser mais facilmente paralelizado. Logo, ao invés de implementar uma função *kernel* e definir a quantidade e a organização de *threads* em blocos, o programador deve apenas identificar quais partes de seu código são paralelizáveis e fazer uso das funções padrão do MATLAB.

4 LINGUAGEM DE COMPUTAÇÃO ABERTA

O OpenCL é um padrão aberto, mantido pelo *Khronos Group*®, que permite o uso de GPUs para desenvolvimento de aplicações paralelas. Ele também permite que os desenvolvedores escrevam códigos de programação heterogêneos, fazendo com que estes programas consigam aproveitar tanto os recursos de processamento das CPUs quanto das GPUs. Além disso, permite programação paralela usando paralelismo de dados e de tarefas (TSUCHIYAMA, 2010).

Apesar de se tratar de um sistema aberto, o *Khronos Group*® é responsável pela padronização de alguns parâmetros. O *Khronos Group*® anunciou recentemente uma versão atualizada do OpenCL. O OpenCL 2.0 é a mais recente evolução do padrão OpenCL, projetado para simplificar ainda mais a programação multiplataforma, permitindo uma variedade de algoritmos e padrões de programação para serem facilmente acelerados (KHRONOS GROUP, 2013). Ainda, poderá fornecer uma série de vantagens interessantes para os desenvolvedores.

O OpenCL também fornece uma linguagem e várias APIs, permitindo que os programadores acelerem aplicações por meio da exploração de paralelismo de dados ou tarefas. Os dispositivos em OpenCL podem ou não compartilhar memória com a CPU e, normalmente, têm um conjunto de instruções de máquina diferente (STONE; GOHARA; SHI, 2010). As APIs fornecidas pelo OpenCL incluem funções para enumerar os dispositivos disponíveis (CPU, GPU e outros aceleradores), gerenciar as alocações de memória, realizar transferências de dados entre CPU e GPU, invocar *kernels* para serem executados nos núcleos da GPU, e verificar erros.

Recentemente, o OpenCL também têm oferecido suporte para CUDA, permitindo o desenvolvimento de aplicativos para serem executados em diversas plataformas. Por meio de sua API, os desenvolvedores podem fazer chamadas a funções *kernels* utilizando um subconjunto limitado da linguagem de programação C em uma GPU (NVIDIA CUDA, 2010). Um programa em OpenCL consiste em *kernels*, que são executados pelo(s) *device*(s), e *host*, que gerencia a execução dos *kernels*. Os *kernels* são executados por *workitens*. Os *workitens* são agrupados em *workgroups*. Os *workgroups* são organizados em um *NDRange*. A Figura 3 mostra a organização dos *workitens* e *workgroups* em um *NDRange*.

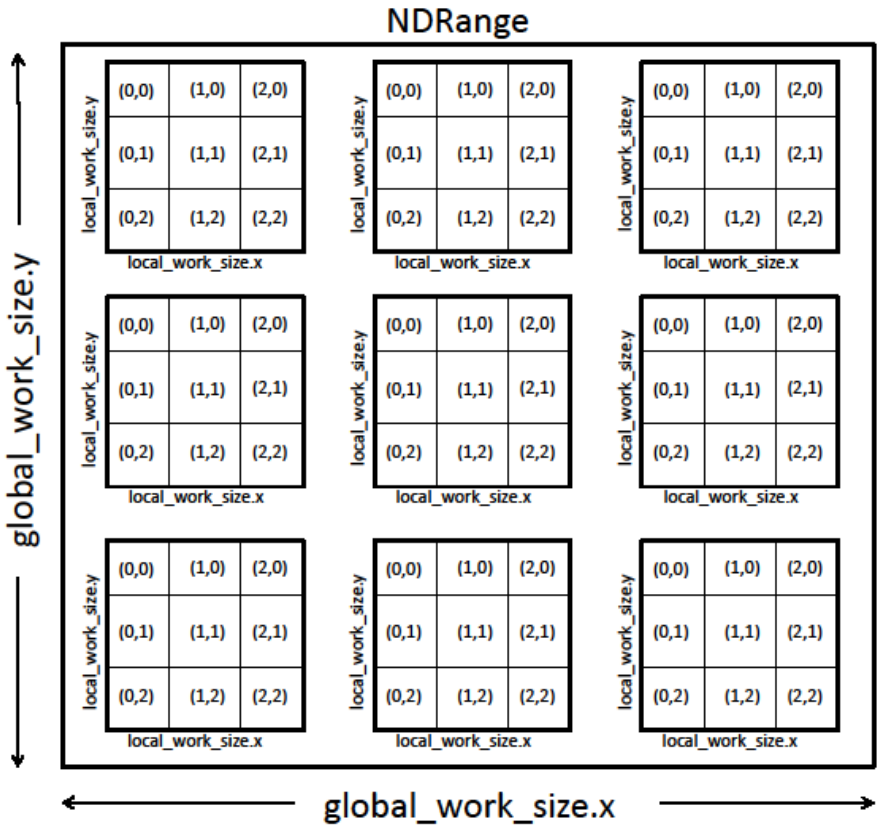

Figura 3 - Organização dos *workgroups* em um *NDRange*
Fonte: Autor

Após seu surgimento, alguns trabalhos têm utilizado o OpenCL na tentativa de aumentar o desempenho computacional de problemas paralelizáveis. Entre eles, pode-se citar o trabalho de Komatsu et al. (2010), que apresentaram uma avaliação de desempenho e portabilidade de programas em OpenCL. Barak et al. (2010) apresentaram algumas aplicações em OpenCL executadas em clusters com muitas GPUs. Grewe e Oboyle (2011) propuseram um esquema de particionamento portátil para programas em OpenCL em sistemas heterogêneos. Bueno, Rodriguez e Sotelino (2013) explorou o potencial computacional de

múltiplas GPUs, utilizando OpenCL, com a finalidade de resolver sistemas de equações lineares de grande porte. Mais recentemente, Suganuma et al. (2013) descreveram suas experiências em programação OpenCL para obter um desempenho escalável para um ambiente de computação heterogênea e distribuída.

Enquanto CUDA é mantido e aprimorado apenas pela NVIDIA®, o OpenCL é suportado por fabricantes como, por exemplo: AMD®, NVIDIA®, APPLE®, INTEL® e IBM®. No entanto, apesar de se tratar de um modelo heterogêneo, permitindo o gerenciamento para portabilidade em multiplataformas, o OpenCL pode se mostrar um tanto quanto complexo em comparação a CUDA. Além disso, independentemente de um código em OpenCL ser suportado por uma grande variedade de dispositvos, isso não significa que o código será executado de forma otimizada em todos eles sem qualquer esforço da parte do programador (KARIMI; DICKSON; HAMZE, 2010).

5 COMPARAÇÃO ENTRE CUDA E OPENCL

CUDA e OpenCL são *frameworks* para programação em GPU. Ambos permitem o uso de GPUs para computação de tarefas de propósito geral que podem ser paralelizadas. CUDA é uma arquitetura proprietária da NVIDIA®, podendo ser utilizada unicamente nas placas gráficas produzidas por esta empresa. O OpenCL é um padrão aberto que pode ser executado em *hardwares* de vários fornecedores.

Alguns programadores afirmam que CUDA é mais "maduro"' (eficiente) e contém APIs de alto nível, as quais são mais convenientes. Entretanto, isso pode mudar haja vista que o OpenCL vem se aprimorando (NVIDIA CUDA, 2010). Uma vantagem do OpenCL é o fato de ele permitir que qualquer fornecedor implemente suporte OpenCL para seus produtos. O modelo de programação em OpenCL é semelhante ao utilizado em CUDA. A Tabela 1 mostra uma comparação entre alguns termos utilizados em CUDA e OpenCL.

Tabela 1 - Comparação entre termos utilizados em CUDA e OpenCL

CUDA	**OpenCL**
Streamming Multiprocessor (SM)	Compute Unit (CU)
Streamming Processor (SP)	Processing Element (PE)
host	*host*
device	*device*
kernel	*kernel*
thread	*workitem*
bloco	*workgroup*
grid	*NDRange*

Fonte: Autor

Alguns trabalhos na literatura realizaram comparações entre CUDA e OpenCL. Por exemplo, Karimi, Dickson e Hamze (2010) apresentaram uma comparação de desempenho computacional entre CUDA e OpenCL. Eles mostraram que, apesar de o OpenCL fornecer um código portátil para execução em diferentes arquiteturas de GPUs, sua generalidade pode implicar em um baixo desempenho. Fang, Varbanescu e Sips (2011) realizaram uma comparação de desempenho abrangente entre CUDA e OpenCL. Os resultados destes autores mostraram que, para a maioria das aplicações, CUDA se mostra, no máximo, 30% melhor do que o OpenCL.

Com base apenas nesses resultados, percebe-se que não há uma conclusão clara sobre qual arquitetura realmente é a melhor. Apenas é possível observar que, enquanto CUDA pode apresentar um bom desempenho computacional para a computação de algumas tarefas, o OpenCL pode se mostrar tão eficiente quanto CUDA.

6 CONCLUSÃO

A utilização da computação paralela vem sendo cada vez mais necessária para o processamento de grandes volumes de dados, contidos em vários tipos de problemas de diversas áreas da ciência. Com isso, a busca pelo aumento de desempenho computacional se torna significante à medida em que o volume de dados aumenta. Não menos importante, a utilização de um bom modelo de programação também se torna necessária para viabilizar o acesso e a computação dos dados. Modelos de programação como CUDA e OpenCL permitem que aplicações possam ser executadas mais facilmente na GPU.

Diversos trabalhos já utilizaram CUDA ou OpenCL para solucionar vários tipos de problemas paralelizáveis. Outros trabalhos apresentaram uma comparação de desempenho computacional entre ambos os modelos. Entretanto, com base na revisão bibliográfica realizada, não foi encontrado algum artigo completo e recente que realizasse uma comparação teórica, destacando claramente qual modelo, de fato, pode ser considerado mais adequado.

O objetivo deste trabalho foi comparar CUDA com o OpenCL apenas em relação à aspectos de *hardware*, *software*, tendências tecnológicas e facilidades de utilização. Apesar de ser uma arquitetura proprietária, foi possível concluir que CUDA, além de ser mais utilizada atualmente e a primogênita, pode ser considerada uma escolha mais viável em comparação com OpenCL.

A escolha do OpenCL pode parecer mais óbvia por ser possível desenvolver programas que poderiam ser executados em qualquer GPU, ao invés de desenvolver uma versão (em CUDA) para placas da NVIDIA®. No entanto, na prática essa escolha pode ser um pouco mais complicada, já que o OpenCL oferece funções e extensões que são específicas para cada família. Além disso, por ter um modelo de gerenciamento para a portabilidade em multiplataformas e multifornecedores, o OpenCL pode ser considerado mais complexo.

Trabalhos futuros poderão realizar comparações mais complexas entre CUDA e OpenCL. Por exemplo, aspectos referentes a execução de instruções em nível de hardware poderão ser analisados. Adicionalmente, possíveis novas arquiteturas e novos modelos de programação poderão surgir e serem investigados para a realização de estudos comparativos.

REFERÊNCIAS

BARAK, A. et al. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In: CLUSTER COMPUTING WORKSHOPS AND POSTERS (CLUSTER WORKSHOPS), 2010, Heraklion. **Proceedings…** Heraklion: IEEE International Conference, 2010. p. 1-7.

BUENO, A. L. C.; RODRIGUEZ, N. R.; SOTELINO, E. D. **Resolução de sistemas de equações lineares de grande porte em clusters multi-GPU utilizando o método do gradiente conjugado em OpenCL**. 2013. Dissertação (Mestrado em Informática) - Departamento de Informática da PUC-Rio, Rio de Janeiro, 2013.

FABRIS, F.; KROHLING, R. A. A co-evolutionary differential evolution algorithm for solving min-max optimization problems implemented on GPU using C-CUDA. **Expert Systems with Applications**, v. 39, n. 12, p. 10324-10333, 2012.

FANG, J.; VARBANESCU, A. L.; SIPS, H. A comprehensive performance comparison of CUDA and OpenCL. In: PARALLEL PROCESSING INTERNATIONAL CONFERENCE (ICPP), 2011, Taipei City. **Proceedings…** Taipei City: [s.n.]. p. 216-225, 2011.

FLYNN, M. J.; RUDD, K. W. Parallel Architectures. **ACM Computing Surveys (CSUR)**, v. 28, n. 1, p. 67-70, 1996.

GAIOSO, R. R. et al. Paralelização do algoritmo Floyd-Warshall usando GPU. In: SIMPÓSIO EM SISTEMAS COMPUTACIONAIS, XIV., 2013, Porto de Galinhas, PE. **Anais...** Porto de Galinhas: UFPE. p. 19-25, 2013.

GREWE, D.; OBOYLE, M. A static task partitioning approach for heterogeneous systems using OpenCL. In: INTERNATIONAL CONFERENCE ON COMPILER CONSTRUCTION: PART OF THE JOINT EUROPEAN CONFERENCES ON THEORY AND PRACTICE OF SOFTWARE, 20th., 2011, Uk. **Proceedings…** Uk: ACM. p. 286-305, 2011.

KARIMI, K.; DICKSON, N.; HAMZE, F. A performance comparison of CUDA and OpenCL. **arXiv preprint arXiv:1005.2581**, 2010.

KHRONOS GROUP. **The open standard for parallel programming of heterogeneous systems**. Disponível em: <http://www.khronos.org/opencl/>. Acesso em: dez. 2013.

KIRK, D. **NVIDIA cuda software and gpu parallel computing architecture**. [S.l.]: NVIDIA Corporation, 2008.

KOMATSU, K. et al. Evaluating performance and portability of OpenCL programs. In INTERNATIONAL WORKSHOP ON AUTOMATIC PERFORMANCE TUNING, 5th., 2010, [S.l.]. **Proceeding…** [S.l.: s.n.], 2010.

KONG, J. et al. Accelerating MATLAB image processing toolbox functions on GPUs. In: WORKSHOP ON GENERAL-PURPOSE COMPUTATION ON GRAPHICS PROCESSING UNITS, 3th., 2010, [S.l.]. **Proceedings…** [S.l.: s.n.]. p. 75-85, 2010.

LITTLE, J; MOLER, C. **MATLAB GPU computing support for NVIDIA CUDA-Enabled GPUs**. The MathWorks, Inc. Disponível em: <http://www.mathworks.com/discovery/matlab-gpu.html>. Acesso em: dez. 2013.

LIU, X.; CHENG, L.; ZHOU, Q. Research and Comparison of CUDA GPU Programming in MATLAB and Mathematica. In: CHINESE INTELLIGENT AUTOMATION CONFERENCE, 2013, [S.l.]. **Proceedings…** [S.l.: s.n.]. p. 251-257, 2013.

NAVAUX, P. O. A. Introdução ao processamento paralelo. **Revista Brasileira de Computação**, v. 5, n. 2, p. 31-43, 1989.

NVIDIA CUDA. **Accelerating MATLAB with CUDA**. NVIDIA Corporation, v. 1, 2007.

NVIDIA CUDA. **NVIDIA CUDA C Programming Best PracticesGuide**. [S.l.]: NVIDIA Corporation, 2009.

_____. **NVIDIA CUDA C ProgrammingGuide**. NVIDIA Corporation, ed. 5.0. [S.l.]: NVIDIA Corporation, 2013.

_____. **OpenCL Programming Guide for the CUDA Architecture**. [S.l.]: NVIDIA Corporation, 2010.

PAULA, L. C. M. et al. Aplicação de Processamento Paralelo em método iterativo para solução de sistemas lineares. In: ENCONTRO ANUAL DE COMPUTAÇÃO, X., 2013a, Catalão. **Anais...** Catalão: UFG. p. 129-136, 2013a.

PAULA, L. C. M. et al. Partial Parallelization of the Successive Projections Algorithm using Compute Unified Device Architecture. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS. 19th., 2013b, [S.l.]. **Proceedings…** [S.l.: s.n.]. p. 737-741, 2013b.

PAULA, L. C. M. Paralelização e comparação de métodos iterativos na solução de sistemas lineares grandes e esparsos. **ForScience**: revista científica do IFMG, v. 1, n. 1, p. 01-12, 2013c.

_____. Implementação Paralela do Método BiCGStab(2) em GPU usando CUDA e Matlab para Solução de Sistemas Lineares. **Revista de Sistemas e Computação**, v. 3, n.2, p. 125-131, 2013d.

_____. Programação Paralela de um Método Iterativo para Solução de Grandes Sistemas de Equações Lineares usando a Integração CUDA-Matlab. **Revista de Sistemas e Computação (Aceito para publicação)**, v. 4, n. 1, 2014a.

PAULA, L. C. M. et al. Parallelization of a Modified Firefly Algorithm using GPU for Variable Selection in a Multivariate Calibration Problem. **International Journal of Natural Computing Research**, v. 4, n. 1, p. 31-42, 2014b.

QUINN, M. J. Parallel Programming. **TMH CSE**, v. 526, 2003.

REESE, J.; ZARANEK, S. **GPU Programming in MATLAB**. The MathWorks, Inc., 2011. Disponível em: <http://www.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html>. Acesso em: dez. 2013.

SIMEK, V.; ASN, R. R. Gpu acceleration of 2d-dwt image compression in matlab with cuda. In: COMPUTER MODELING AND SIMULATION, EMS'08. SECOND UKSIM

EUROPEAN SYMPOSIUM, 2008, Liverpool. **Proceedings…** Liverpool: [s.n.]. p. 274-277, 2008.

SMITH, S. M. **The GPU Computing Revolution**. [S.l.: s.n.] 2011.

STALLINGS, W. **Arquitetura e organização de computadores**. 5. ed. São Paulo: Prentice Hall, 2002.

STONE, J.; GOHARA, D.; SHI, G. OpenCL: a parallel programming standard for heterogeneous computing systems. **Computing in science & engineering**, v. 12, n. 3, p. 66, 2010.

SUGANUMA, T. et al. Scaling analytics applications with OpenCL for loosely coupled heterogeneous clusters. In: ACM INTERNATIONAL CONFERENCE ON COMPUTING FRONTIERS, 2013, New York. **Proceedings…** New York: ACM. p. 35, 2013.

TSUCHIYAMA, R. **The OpenCL Programming Book**. [S.l.]: Fixstars, 2010.

YLDIRIM, A. A.; OZDOGAN, C. Parallel wavelet-based clustering algorithm on GPUs using CUDA. **Procedia Computer Science**, v. 3, p. 396-400, 2011.

Recebido em: 05/12/2013

Aprovado em: 04/03/14