Uma Comparação Teórica entre dois Modelos de Programação Paralela em GPU

Lauro Cássio Martins de Paula[1]

[1] Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo (USP)
CEP: 13.566-590 – São Carlos – SP – Brazil

laurocassio21@gmail.com

Abstract. *This paper presents a comparison between two parallel architectures: Compute Unified Device Architecture (CUDA) e 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. Thus, 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 performance 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.*

Resumo. *Apresenta-se neste trabalho uma comparação entre dois modelos utilizados para programação paralela: Compute Unified Device Architecture (CUDA) e Open Computing Language (OpenCL). Alguns trabalhos na literatura apresentaram uma comparação de desempenho computacional entre esses dois modelos. Entretanto, ainda não existe algum artigo recente e completo que destaca claramente qual modelo, de fato, pode ser considerado o mais eficiente. Portanto, o objetivo deste trabalho é realizar uma comparação apenas em nível de $hardware$, $software$, tendências tecnológicas e facilidades de utilização, evidenciando aquele que pode apresentar o melhor desempenho de uma maneira geral. Para tal, descreve-se os principais trabalhos que já fizeram uso de pelo menos um dos modelos. 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 utilizado apenas nas placas gráficas da $NVIDIA^{®}$, tem sido uma referência e mais utilizado ultimamente.*

1. Introdução

A Computação Paralela (CP) tem contribuído bastante para diversas áreas da ciência, que vão desde simulações computacionais a aplicações científicas. A CP é uma forma eficiente do processamento da informação, com ênfase na exploração de eventos concorrentes no processo computacional [Smith 2011]. 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 [$CUDA^{TM}$ 2013].

Atualmente têm-se desenvolvido dois tipos de processadores: $Multi - core$ e $Many - core$. Os processadores $Multi - core$, normalmente, contém poucos núcleos, porém com um grande poder de processamento [Stallings 2002]. 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 ($threads$) [Smith 2011]. Os $Many - core$ são desenvolvidos com dezenas ou centenas de núcleos mais simples, otimizados para uma maior vazão na execução de instruções, executando centenas ou até mesmo milhares de $threads$ [Kirk and Hwu 2011]. As *Graphics Processing Units* (GPU) são exemplo de processadores $Many - core$ [Paula 2013].

De forma correspondente à evolução do $hardware$, novos modelos de programação paralela têm sido elaborados, destacando-se dois deles: *Compute Unified Device Architecture* (CUDA) [$CUDA^{TM}$ 2013] e *Open Computing Language* (OpenCL) [Tsuchiyama et al. 2010]. Tais modelos, devido a uma ampla disponibilidade de *Application Programming Interfaces* (API), permitem que aplicações possam ser executadas mais facilmente na GPU [Gaioso et al. 2013].

Muitos trabalhos na literatura têm utilizado CUDA e (ou) OpenCL para a solução de diversos tipos de problemas paralelizáveis. Entretanto, uma comparação teórica e tecnológica entre ambos os modelos ainda tem sido pouco investigada. 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 para a maioria das aplicações [Karimi et al. 2010]. Por outro lado, outros evidenciam que o OpenCL pode ser uma boa alternativa em relação a CUDA [Fang et al. 2011]. 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. Foi possível concluir que CUDA, embora seja uma tecnologia proprietária, tem sido uma referência e pode ser considerada mais eficiente.

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 o OpenCL. Por fim, a Seção 5 mostra as conclusões do trabalho.

2. Unidade de Processamento Gráfico

As GPUs foram inicialmente desenvolvidas como uma tecnologia orientada à vazão, otimizada para cálculos de uso itensivo de dados, onde muitas operações idênticas podem ser realizadas em paralelo sobre diferentes dados (*Single Instruction Multiple Data - SIMD*) [Paula et al. 2013a]. Diferente de uma CPU $multicore$, a qual executa algumas $threads$ em paralelo, a GPU foi projetada para executar milhares de $threads$ [Paula et al. 2014].

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. 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 and Hwu 2011].

Por outro lado, como toda tecnologia, as GPUs possuem suas limitações. Dependendo do volume de dados, o desempenho computacional da GPU pode se mostrar inferior quando comparado ao desempenho da CPU [Paula et al. 2013a]. Isso implica que a quantidade de dados a serem transferidos para a memória da GPU deve ser levado em consideração, devido à existência de um *overhead* associado à paralelização das tarefas na GPU [$CUDA^{TM}$ 2009a], [Gaioso et al. 2013]. Fatores em relação ao tempo de acesso em memória também podem influenciar no desempenho computacional. Ou seja, o acesso à memória global da GPU geralmente apresenta uma alta latência e pode estar sujeito a um acesso aglutinado aos dados em memória [$CUDA^{TM}$ 2009a].

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 de programação 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 [$CUDA^{TM}$ 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 [$CUDA^{TM}$ 2009a]. 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), onde algumas palavras-chave são utilizadas para rotular as funções paralelas (*kernels*) e suas estruturas de dados [Kirk and Hwu 2011].

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 disparado, ele é executado como uma grade (*grid*) de *threads* paralelas [$CUDA^{TM}$ 2013]. Como ilustra a Figura 1, as *threads* em um *grid* são organizadas em uma hierarquia de dois níveis, onde cada *grid* consiste em um ou mais blocos de *threads*.

Desde o seu surgimento, alguns trabalhos têm utilizado CUDA para a paralelização de vários tipos de problemas. Por exemplo, Paula *et al.* (2013b) utilizaram CUDA-C para paralelizar o método BiCGStab(2), utilizado para solução de sistemas lineares. Paula *et al.* (2013a) propuseram uma estratégia de paralelização para a fase 2 do Algoritmo das Projeções Sucessivas utilizando CUDA-C. Paula (2013) utilizou a estratégia proposta em [Paula et al. 2013b] e apresentou uma comparação entre métodos iterativos na solução de sistemas lineares grandes e esparsos. Por fim, 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 em um grafo.

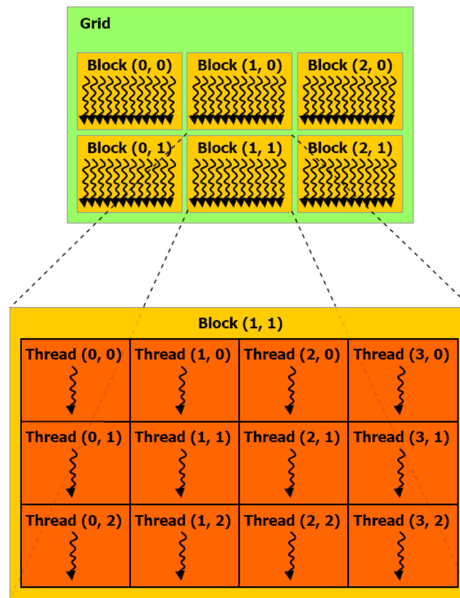

Figura 1. $Grid$ **com vários blocos de** $threads$ **[**$CUDA^{TM}$ **2013].**

Recentemente, a $MathWorks^{®}$ [Little and Moler 2013] 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 and 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 sobracarregados ($GPUArray$). 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 and 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 et al. 2013]. Isso ocorre porque os aspectos de exploração de paralelismo são realizados pelo próprio PCT [Little and Moler 2013]. 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 [$CUDA^{TM}$ 2007]. Simek e Asn (2008) apresentaram uma implementação em MATLAB com CUDA para compressão de imagens médicas [Simek and Asn 2008]. Mais recentemente, Liu *et al.* [Liu et al. 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

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 et al. 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, que é a mais recente evolução do padrão OpenCL, projetado para simplificar ainda mais a programação multi-plataforma, permitindo uma variedade de algoritmos e padrões de programação para serem facilmente acelerados [Khronos 2013]. Os dispositivos em OpenCL podem ou não compartilhar memória com a CPU e, normalmente, têm um conjunto de instruções de máquina diferente [Stone et al. 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, disparar $kernels$, 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 [$CUDA^{TM}$ 2009b]. Ainda, por meio de suas diversas APIs, os desenvolvedores podem fazer chamadas à funções $kernels$ utilizando um subconjunto limitado da linguagem de programação C. 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 2 mostra a organização dos $workitens$ e $workgroups$ em um $NDRange$.

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 [Komatsu et al. 2010]. Barak *et al.* (2010) apresentaram algumas aplicações em OpenCL executadas em $clusters$ com muitas GPUs [Barak et al. 2010]. 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 [Suganuma et al. 2013].

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 mos-

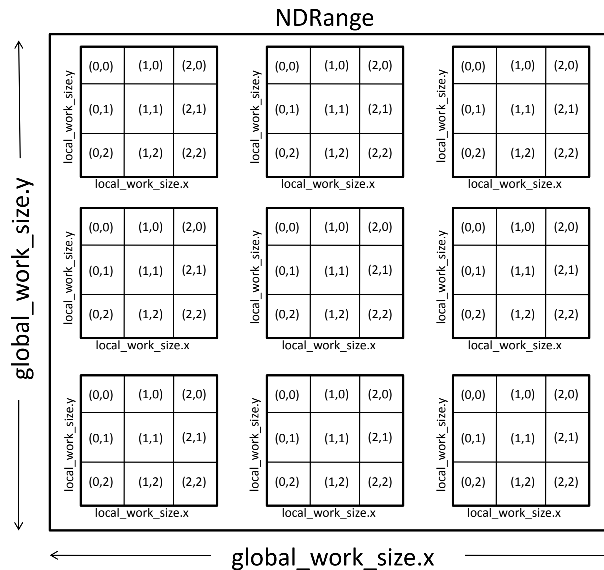

Figura 2. Organização dos $workgroups$ **e** $workitens$ **em um** $NDRange$**.**

trar 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 [$CUDA^{TM}$ 2009b].

5. Conclusões

A utilização da computação paralela vêm 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 do 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 necessário 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. Por exemplo, Karimi *et al.* (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 [Karimi et al. 2010]. Por outro lado, Fang *et al.* (2011) realizaram uma comparação de desempenho abrangente entre CUDA e OpenCL. Os resultados deles mostraram que, para a maioria das aplicações, CUDA se mostra, no máximo, 30% melhor do que o OpenCL [Fang et al. 2011]. Considerando apenas esses resultados, percebe-se que não há uma conclusão clara sobre qual arquitetura realmente é mais eficaz. 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.

Com base na revisão bibliográfica realizada, não foi encontrado algum artigo com-

pleto e recente que realiza uma comparação teórica, destacando claramente qual modelo, de fato, pode ser considerado mais adequado. Portanto, 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 tecnologia proprietária, foi possível concluir que CUDA, além de ser o primogênito e considerado mais "maduro", pode ser 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 execução apenas nas placas da $NVIDIA^{®}$. Entretanto, na prática essa escolha pode ser um pouco mais complicada, já que o OpenCL oferece funções e extensões que são específicas para cada família [$CUDA^{TM}$ 2009b]. Ainda, por ter um modelo de gerenciamento para a portabilidade em multiplataformas e multifornecedores, o OpenCL pode ser considerado mais complexo [Kirk and Hwu 2011].

Trabalhos futuros poderão realizar comparações mais complexas entre CUDA e OpenCL. Por exemplo, aspectos como execução de instruções à 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., Ben-Nun, T., Levy, E., and Shiloh, A. (2010). A package for opencl based heterogeneous computing on clusters with many gpu devices. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–7. IEEE.

$CUDA^{TM}$, N. (2007). *Accelerating MATLAB with CUDA*, volume 1. NVIDIA Corporation.

$CUDA^{TM}$, N. (2009a). *NVIDIA CUDA C Programming Best Practices Guide*. NVIDIA Corporation, 2701 San Tomas Expressway Santa Clara, CA 95050.

$CUDA^{TM}$, N. (2009b). *OpenCL Programming Guide for the CUDA Architecture*. NVIDIA Corporation.

$CUDA^{TM}$, N. (2013). *NVIDIA CUDA C Programming Guide*. NVIDIA Corporation, 2701 San Tomas Expressway Santa Clara, CA 95050, 5.0 edition.

Fang, J., Varbanescu, A. L., and Sips, H. (2011). A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE.

Gaioso, R. D. R. A., Jradi, W., de Paula, L. C. M., Alencar, W., Martins, W. S., Nascimento, H., and Caceres, E. (2013). Paralelização do algoritmo floyd-warshall usando gpu. In *XIV Simposio em Sistemas Computacionais*, pages 19–25.

Karimi, K., Dickson, N. G., and Hamze, F. (2010). A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*.

Khronos, G. (2013). The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

Kirk, D. B. and Hwu, W. (2011). *Programando para Processadores Paralelos*. Elsevier, 1 edition.

Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H., and Kobayashi, H. (2010). Evaluating performance and portability of opencl programs. In *The fifth international workshop on automatic performance tuning*.

Little, J. and Moler, C. (2013). Matlab gpu computing support for nvidia cuda-enabled gpus. http://www.mathworks.com/discovery/matlab-gpu.html.

Liu, X., Cheng, L., and Zhou, Q. (2013). Research and comparison of cuda gpu programming in matlab and mathematica. In *Proceedings of 2013 Chinese Intelligent Automation Conference*, pages 251–257. Springer.

Paula, L. C. M. (2013). Paralelização e comparação de métodos iterativos na solução de sistemas lineares grandes e esparsos. *ForScience: Revista Científica do IFMG*, 1(1):01–12.

Paula, L. C. M., Soares, A. S., Lima, T. W., Delbem, A. C. B., Coelho, C. J., and Filho, A. R. G. (2014). Parallelization of a modified firefly algorithm using gpu for variable selection in a multivariate calibration problem. *International Journal of Natural Computing Research*, 4(1):31–42.

Paula, L. C. M., Soares, A. S., Soares, T. W., Martins, W. S., Filho, A. R. G., and Coelho, C. J. (2013a). Partial parallelization of the successive projections algorithm using compute unified device architecture. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 737–741.

Paula, L. C. M., Souza, L. B. S., Souza, L. B. S., and Martins, W. S. (2013b). Aplicação de processamento paralelo em método iterativo para solução de sistemas lineares. In *X Encontro Anual de Computação*, pages 129–136.

Reese, J. and Zaranek, S. (2011). Gpu programming in matlab. http://www.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html.

Simek, V. and Asn, R. R. (2008). Gpu acceleration of 2d-dwt image compression in matlab with cuda. In *Computer Modeling and Simulation, 2008. EMS'08. Second UKSIM European Symposium on*, pages 274–277. IEEE.

Smith, S. M. (2011). The gpu computing revolution.

Stallings, W. (2002). *Arquitetura e Organização de Computadores*. Prentice Hall, São Paulo, SP, Brasil, 5 edition.

Stone, J. E., Gohara, D., and Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66.

Suganuma, T., Krishnamurthy, R. B., Ohara, M., and Nakatani, T. (2013). Scaling analytics applications with opencl for loosely coupled heterogeneous clusters. In *Proceedings of the ACM International Conference on Computing Frontiers*, page 35. ACM.

Tsuchiyama, R., Nakamura, T., Iizuka, T., Asahara, A., Son, J., and Miki, S. (2010). *The OpenCL Programming Book*. Fixstars.