Programação Paralela Híbrida em CPU e GPU: Uma Alternativa na Busca por Desempenho

André Luís Stefanello[1], Crístian Cleder Machado[1], Dioni da Rosa[1], Maurício Sulzbach[1], Rodrigo Wilhelm Moerschbacher[1], Thiago Roberto Sarturi[1]

[1]Departamento de Engenharias de Ciência da Computação – Universidade Regional Integrada do Alto Uruguai e das Missões – Campus de Frederico Westphalen (URI)
Caixa Postal 184 – 98.400-000 – Frederico Westphalen – RS – Brasil

{andres, cristian, dioni, sulzbach, inf17343, tsarturi}@uri.edu.br

Abstract. *Currently through technological advancement, the electronics industry has provided a number of new devices with computational characteristics, such as the GPU. This evolution, ally with new parallel programming APIs have open new options in high-performance computing. In addition, join more than a device together with computational power, such as eg, CPU and GPU in a hybrid environment tends to have performance gains. Thus, this article presents a performance evaluation of hybrids parallels algorithms for CPU and CPU through the APIs MPI, OpenMP, CUDA and OpenCL. Were developed and tested several variations of calculating the hybrid scalar product in CPU and GPU and found that the hybrid parallel programming is an interesting alternative in the search by a better performance.*

Resumo. *Atualmente através do avanço tecnológico, a indústria de eletrônicos tem disponibilizado uma série de novos dispositivos com características computacionais, como é o caso da GPU. Essa evolução, aliada a novas APIs de programação paralela tem aberto novas alternativas na computação de alto desempenho. Além disso, unir mais de um dispositivo com poder computacional, como por exemplo, CPU e GPU em um ambiente híbrido tende a ter ganhos de desempenho. Sendo assim, o presente artigo objetiva apresentar uma avaliação de desempenho de algoritmos paralelos híbridos para CPU e CPU através das APIs MPI, OpenMP, CUDA e OpenCL. Foram desenvolvidas e testadas diversas variações do cálculo do produto escalar híbrido em CPU e GPU e constatado que a programação paralela híbrida é uma alternativa interessante na busca por um melhor desempenho.*

1. Introdução

A programação de alto desempenho tem apresentado um crescimento intenso nos últimos tempos, visto que dispositivos como a *Central Processing Unit* (CPU) e a *Graphics Processing Unit* (GPU) evoluíram muito na sua capacidade computacional. A CPU, por exemplo, em contrapartida a estagnação da velocidade do *clock* teve a introdução de mais núcleos dentro de um único processador. Já a GPU passou de um simples processador gráfico para um coprocessador paralelo, capaz de executar milhares de operações simultaneamente, gerando uma grande capacidade computacional, que em muitas vezes supera o poder de processamento das tradicionais CPUs. Aliado a isso, o surgimento de algumas APIs (*Application Programming Interface*) com suporte ao paralelismo possibilitaram usufruir da real capacidade computacional desses dispositivos. Destacam-se OpenMP (*Open Multi Processing*) e MPI (*Message-Passing Interface*), que são utilizadas para paralelizar tarefas à CPU e CUDA (*Compute Unified*

Device Architecture) e OpenCL (*Open Computing Language*) para execução paralela em GPU.

Buscando aumentar a capacidade de processamento paralelo aliado a um custo acessível, tem-se nos últimos anos um crescimento do desenvolvimento de algoritmos paralelos de forma híbrida utilizando CPU e GPU. A programação paralela híbrida objetiva unir dispositivos de computação com diferentes arquiteturas trabalhando em conjunto, visando atingir um maior desempenho. Além disso, a programação híbrida busca fazer com que cada conjunto de instruções possa ser executado na arquitetura que melhor se adapte. Dessa forma, o presente trabalho irá se concentrar no desenvolvimento de aplicações híbridas para CPU e GPU utilizando OpenMP, MPI, CUDA e OpenCL. Foram desenvolvidos ao longo desta pesquisa, diversos algoritmos para ao cálculo do produto escalar, com variações de carga, mesclando APIs para CPU e GPU. Por fim, foram realizadas as análises de desempenho e eficiência dos algoritmos desenvolvidos e constatado que a programação paralela híbrida é uma alternativa interessante na busca por um melhor desempenho.

O presente trabalho está estruturado da seguinte forma. Na seção 2 é abordada a programação paralela híbrida. A sessão 3 trata das principais APIs para o desenvolvimento de aplicativos paralelos para CPU e GPU. A proposta de algoritmos paralelos híbridos é detalhada na sessão 4, bem como uma análise sobre o desempenho e eficiência de cada um deles é debatida na sessão 5. Para finalizar, a sessão 6 trás as conclusões deste trabalho.

2. Programação Paralela Híbrida

O principal objetivo da programação paralela híbrida é tirar proveito das melhores características dos diferentes modelos de programação visando alcançar um melhor desempenho. Ela consiste em mesclar a paralelização de tarefas complexas ou com intensas repetições, com atividades mais simples em mais de uma unidade processadora [Ribeiro e Fernandes 2013] ou ainda, distribuir a paralelização de acordo com as características computacionais dessas unidades. Ao trabalhar com paralelismo híbrido é importante considerar também como cada paradigma/API pode paralelizar o problema e como combiná-los de modo a alcançar o melhor resultado [Silva 2006].

A arquitetura do *hardware* que será utilizada deve ser sempre levada em consideração no momento que se projeta uma aplicação híbrida [Ribeiro e Fernandes 2013]. Esse fato atualmente ganha mais força pelo surgimento de uma gama de dispositivos com capacidades computacionais, porém de arquiteturas diferentes. Segundo Silva [2006], o desenvolvimento de aplicações utilizando mais de um modelo de programação gera benefícios como a possibilidade de balanceamento de carga da aplicação e o poder computacional que as unidades processadoras em conjunto oferecem. Em contra partida, em algumas oportunidades há a necessidade de replicação de dados e sincronização de tarefas que acabam influenciando no tempo de execução. Existem atualmente diversas APIs de programação paralela, destacando-se OpenMP e MPI para execução em CPU, CUDA e OpenCL para processamento em GPU. Alguns conceitos dessas APIs serão abordados a seguir.

3. APIs de Programação Paralela

As principais características das APIs de programação paralela OpenMP e MPI para CPU e CUDA e OpenCL para GPU são apresentadas a seguir.

3.1. OpenMP

OpenMP (*Open Multi Processing*) é uma especificação que fornece um modelo de programação paralela com compartilhamento de memória. Essa API é composta por um conjunto de diretivas que são adicionadas as linguagens C/C++ e Fortran [OpenMP 2013] utilizando o conceito de *threads*, porém sem que o programador tenha que trabalhar diretamente com elas [Matloff 2013]. Esse conjunto de diretivas quando acionado e adequadamente configurado cria blocos de paralelização e distribui o processamento entre os núcleos disponíveis. O programador não necessita se preocupar em criar *threads* e dividir as tarefas manualmente no código fonte. O OpenMP se encarrega de fazer isso em alto nível.

Essa API representa um padrão que define como os compiladores devem gerar códigos paralelos através da incorporação nos programas sequenciais de diretivas que indicam como o trabalho será dividido entre os *cores*. Dessa forma, muitas aplicações podem tirar proveito desse padrão com pequenas modificações no código fonte. No OpenMP, a paralelização é realizada com múltiplas *threads* dentro de um mesmo processo. As *threads* são responsáveis por dividir o processo em duas ou mais tarefas que poderão ser executadas simultaneamente [Sena e Costa 2008].

3.2. MPI

MPI (*Message-Passing Interface*) é uma biblioteca desenvolvida para ambientes de memória distribuída, máquinas paralelas massivas e redes heterogêneas. Oferece um conjunto de rotinas para o desenvolvimento de aplicações através de troca de mensagens, possibilitando que os processos se comuniquem [MPI 2012]. O funcionamento do MPI é de certa forma simples. Os problemas são divididos em partes menores e essas são distribuídas para que os processadores as executem. Os resultados obtidos pelos processadores são enviados a um processador receptor que coleta os dados, agrupa e fornece o resultado esperado. A divisão das tarefas e a distribuição aos processadores são atribuições do programador, ou seja, no MPI o paralelismo é explícito.

O objetivo do MPI é construir um padrão para desenvolver programas que envolvam a transmissão de mensagens (*send and recieve*). Esse padrão preocupa-se com a praticidade, portabilidade, eficiência e flexibilidade na troca de mensagens. Apresenta como principais características a comunicação eficiente, evitando cópia da memória de um processador para a memória de outro processador, interface com as linguagens C/C++ e Fortran, sendo que a semântica da inferface é independente da linguagem e possibilidade de implementação em diferentes plataformas [MPI 2012].

3.3. CUDA

CUDA (*Compute Unified Device Architecture*) é uma plataforma de computação paralela e um modelo de programação criados pela NVIDIA em 2006. Seu objetivo é possibilitar ganhos significativos de performance computacional aproveitando os recursos das unidades de processamento gráfico (GPU) [NVIDIA 2013]. A tecnologia CUDA é de abordagem proprietária, concebida para permitir acesso direto ao *hardware* gráfico específico da NVIDIA. Ao utilizar CUDA também é possível gerar código tanto para a CPU como para a GPU. Em CUDA, a GPU é vista como um dispositivo de computação adequado para aplicações paralelas. Tem seu próprio dispositivo de memória de acesso aleatório e pode ser executado através de um número muito elevado de *threads* em paralelo.

Na arquitetura CUDA a GPU é implementada como um conjunto de multiprocessadores. Cada um dos multiprocessadores têm várias *Arithmetic Logic Unit* (ALU) que em qualquer ciclo de *clock* executam as mesmas instruções, mas em dados diferentes. As ALUs podem acessar através de leitura e escrita a memória compartilhada do multiprocessador e a memória RAM (*Random Access Memory*) do dispositivo [Manavski 2013].

3.4. OpenCL

OpenCL (*Open Computing Language*) é uma API independente de plataforma que permite aproveitar as arquiteturas de computação paralelas disponíveis, como CPUs *multi-core* e GPUs, tendo sido desenvolvida objetivando a portabilidade. Essa API, criada pelo Khronos Group em 2008, define uma especificação aberta em que os fornecedores de *hardware* podem implementar. Por ser independente de plataforma, a programação em OpenCL é mais complexa se comparada a uma API específica, como é o caso da CUDA [SANTOS, et al. 2013]. Enquanto a arquitetura CUDA está restrita aos dispositivos fabricados pela NVIDIA, o OpenCL possui um amplo suporte de fabricantes, bastando apenas a adequação de seu SDK (*Software Development Kit*) ao *framework* [SANTOS, et al. 2013].

Atualmente na sua versão 2.0 [Khronos Group, 2013], a especificação OpenCL é realizada em três partes: linguagem, camada de plataforma e *runtime*. A especificação da linguagem descreve a sintaxe e a API para escrita de código em OpenCL. A camada de plataforma fornece ao desenvolvedor acesso às rotinas que buscam o número e os tipos de dispositivos no sistema. Já o *runtime* possibilita ao desenvolvedor enfileirar comandos para execução nos dispositivos, sendo também o responsável por gerenciar os recursos de memória e computação disponíveis. Na próxima sessão será apresentada a proposta de algoritmo paralelo híbrido adotada neste trabalho.

4. Proposta de Algoritmo Paralelo Híbrido

Para o presente trabalho foram desenvolvidos algoritmos híbridos que mesclam instruções em CPU e GPU. O cálculo do produto escalar foi o problema escolhido para a aplicação dos algoritmos híbridos, com diferentes cargas de trabalho e divisão de tarefas. Primeiramente foi desenvolvido um algoritmo para o cálculo do produto escalar na linguagem C, de forma sequencial, com o objetivo de definir o tempo que a aplicação têm para realizar o cálculo e assim saber se o desempenho dos algoritmos híbridos é satisfatório ou não.

Na sequencia foram desenvolvidos 4 algoritmos utilizando OpenMP, MPI, CUDA e OpenCL, também para o cálculo do produto escalar, sempre obedecendo a utilização de uma API para CPU e outra para GPU, com as seguintes características:

a) Algoritmo híbrido utilizando OpenMP e CUDA denominado OMP/CUDA;

b) Algoritmo híbrido utilizando OpenMP e OpenCL denominado OMP/OCL;

c) Algoritmo híbrido utilizando MPI e CUDA denominado MPI/CUDA e

d) Algoritmo híbrido utilizando MPI e OpenCL denominado MPI/OCL.

5. Resultados Obtidos

Os algoritmos desenvolvidos nesse trabalho foram testados com diferentes tamanhos de cargas, a dizer, tamanho dos vetores e número de repetições. Em todas as variações realizadas, o tempo de execução e o *speedup* dos algoritmos mantiveram-se proporcionais a carga nele alocada. O gráfico da figura 1 mostra o *speedup* dos quatro algoritmos híbridos em relação ao algoritmo sequencial. Nesse teste foram submetidos em cada um dos vetores 20.000.000 de registros, sendo executados repetidamente 1.000 vezes. A média de 10 execuções foi o parâmetro utilizado para o tempo e para o *speedup*. Para os algoritmos que utilizavam as APIs OpenMP e MPI foram definidas 4 *threads* e 4 processos executores, respectivamente.

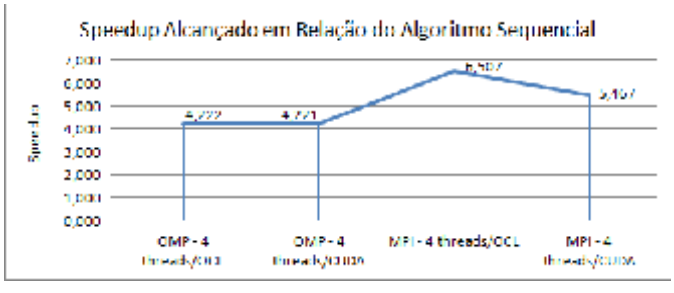

Figura 2. *Speedup* alcançado em relação ao algoritmo sequencial

Como o algoritmo híbrido MPI e OpenCL obteve o menor tempo de execução, consequentemente resultou em um *speedup* melhor, se coparado aos outros algoritmos. A divisão da carga de trabalho foi de 50% para CPU e 50% para GPU. Apesar das execuções híbridas apresentarem um desempenho satisfatório em relação à execução serial, em nenhum dos testes realizados uma das aplicações híbridas, com divisão igualitária de carga de trabalho obteve *speedup* igual ou superior a uma aplicação desenvolvida através de uma única API. Objetivando investigar os motivos que levaram os algoritmos híbridos a não obtenção de um *speedup* próximo dos algoritmos paralelizados em apenas uma API, foi realizado um estudo sobre a real capacidade de processamento de cada dispositivo (CPU e GPU). Dessa forma, configuraram-se cargas de execução de acordo com a capacidade de processamento de cada dispositivo.

5.1 Reconfiguração das Cargas de Trabalho

Observou-se através dos testes realizados que o tempo de transferência de dados da memória da CPU para a memória da GPU é pequeno, aproximadamente 1,5 segundos para uma divisão de carga de trabalho. Também, a capacidade de processamento da GPU é maior que a CPU, devido a quantidade de núcleos executores. Sendo assim, realizou-se a reconfiguração das cargas de trabalho para adequar o volume de instruções a capacidade de execução dos dispositivos. Os algoritmos híbridos sofreram as variações de cargas de trabalho (forma empírica), demonstradas pela tabela 1.

Tabela 1. Cargas de trabalho atribuídas aos algoritmos híbridos

Teste	Carga CPU	Carga GPU
1	50% do trabalho	50% do trabalho
2	30% do trabalho	70% do trabalho
3	20% do trabalho	80% do trabalho
4	15% do trabalho	85% do trabalho
5	10% do trabalho	90% do trabalho
6	5% do trabalho	95% do trabalho

O *speedup* atingido pelo algoritmo híbrido OpenMP e CUDA é ilustrado pela figura 2 e demonstra que quanto maior a carga de trabalho para GPU, menor é o tempo de execução do algoritmo. A carga do algoritmo híbrido para atingir um *speedup* semelhante ao algoritmo desenvolvido somente em CUDA é de 5% para OpenMP e 95% para CUDA. Mesmo com o aumento significativo da carga de trabalho o algoritmo híbrido OpenMP e CUDA não superou o *speedup* desenvolvido pelo algoritmo CUDA.

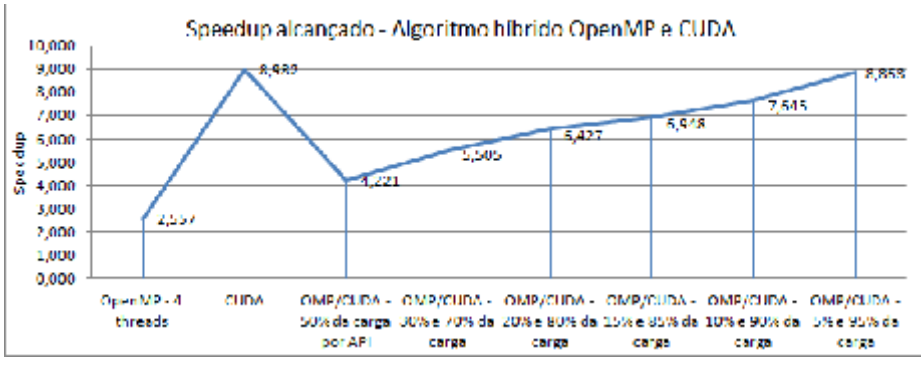

Figura 3. *Speedup* alcançado pelo algoritmo híbrido OpenMP e CUDA

O algoritmo híbrido OpenMP e OpenCL também foi reconfigurado, utilizando as mesmas cargas de trabalho, conforme apresentadas pela tabela 1. Para superar a aceleração produzida pelo algoritmo em OpenCL, a carga de trabalho do algoritmo híbrido OpenMP e OpenCL foi de 10% para CPU e 90% para GPU, obtendo assim um *speedup* de 12,346. O ápice da aceleração atingida foi de 16,727 quando as cargas foram configuradas com 5% do trabalho para OpenMP e 95% do trabalho para OpenCL, conforme ilustra a figura 3.

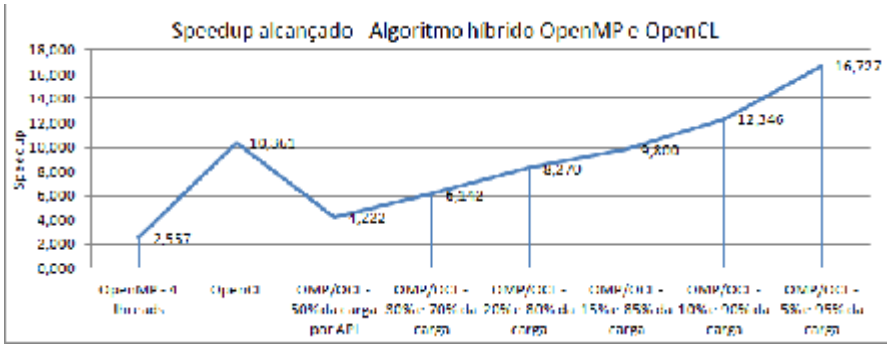

Figura 4. *Speedup* alcançado pelo algoritmo híbrido OpenMP e OpenCL

A reconfiguração das cargas de trabalho do algoritmo híbrido MPI e CUDA possibilitou que o *speedup* atingido fosse semelhante ao algoritmo desenvolvido em CUDA. Para atingir esse desempenho o algoritmo híbrido MPI e CUDA teve a seguinte distribuição de cargas de trabalho: 5% para CPU e 95% para GPU. A figura 4 ilustra o *speedup* alcançado pelo algoritmo híbrido MPI e CUDA.

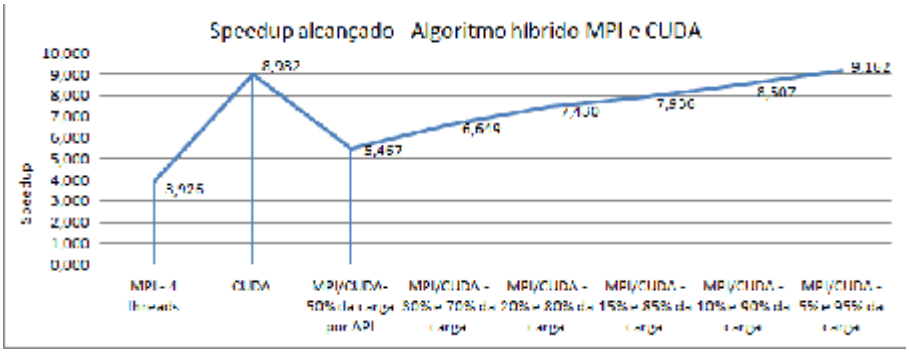

Figura 5. *Speedup* alcançado pelo algoritmo híbrido MPI e CUDA

A diferença de desempenho mais significativa ocorreu na redistribuição das cargas de trabalho no algoritmo híbrido MPI e OpenCL. Com 20% da carga para CPU e 80% da carga para GPU foi o suficiente para superar o *speedup* de 10,361 do algoritmo desenvolvido em OpenCL. Nessa configuração o algoritmo híbrido MPI e OpenCL atingiu o *speedup* de 11,527. A cada nova reconfiguração de carga, quanto mais instruções e dados para serem executados em GPU, maior foi o desempenho do algoritmo híbrido. De acordo com a figura 5, com 5% da carga para CPU e 95% da carga de trabalho para GPU obteve-se *speedup* de 18,860 para o algoritmo híbrido MPI e OpenCL, o melhor desempenho obtido neste trabalho.

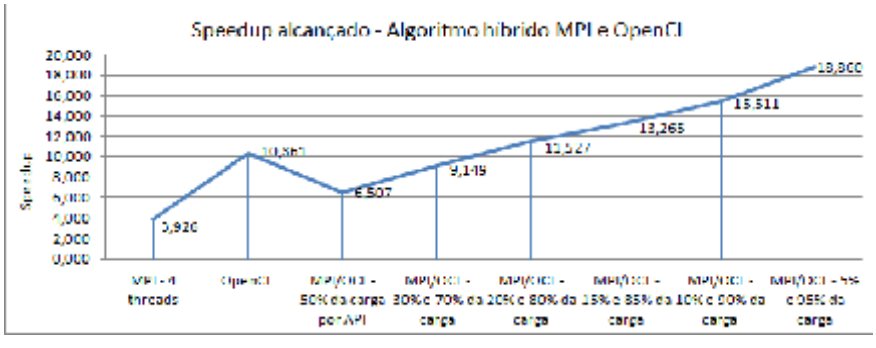

Figura 6. *Speedup* alcançado pelo algoritmo híbrido MPI e OpenCL

5.2 Ambientes de compilação e execução

Os algoritmos desenvolvidos neste trabalho foram compilados através do *nvcc* 4.2*, gcc* 4.6.3 e *mpicc* 4.6.3. Foram utilizadas as APIs OpenMP 4.7.2, MPI MPICH2 1.4.1, CUDA 4.2.1 e OpenCL 1.1. O ambiente de execução foi um servidor com processador Intel Xeon de 2.4GHz e 12 GB de memória RAM, na distribuição Linux Debian 6.0. A GPU foi o modelo NVIDIA Tesla M2050, com 448 núcleos e 3 GB de memória.

6. Conclusão

Buscou-se neste trabalho apresentar a programação paralela híbrida em CPU e GPU como uma alternativa para ganho de desempenho através da utilização das APIs OpenMP, MPI, CUDA e OpenCL. Constatou-se que nos algoritmos híbridos, uma divisão igualitária das instruções e dados entre CPU e GPU resultou em um *speedup* considerável, se comparado ao algoritmo sequencial, porém em se tratando de algoritmos paralelizados em somente uma das APIs obteve-se desempenho inferior. Analisando a arquitetura dos dispositivos utilizados, averiguou-se que através da grande quantidade de núcleos executores da GPU deveria ocorrer uma melhor redistribuição das instruções. Ao realizar a redistribuição das cargas de trabalho observou-se que os

algoritmos híbridos para CPU e GPU tendem a serem mais rápidos que os algoritmos paralelos desenvolvidos em uma única API.

Através dos resultados obtidos ficou evidenciada que a paralelização híbrida é uma boa alternativa para ganho de desempenho. Utilizando os recursos de CPU e GPU unidos e distribuindo as instruções de acordo com a capacidade de processamento de cada dispositivo pode-se obter um tempo menor na execução de tarefas. Também se pode observar que os algoritmos desenvolvidos somente em GPU obtiveram um desempenho melhor, se comparados aos algoritmos desenvolvidos utilizando paralelização para CPU. Isso comprova a grande ascensão dos dispositivos gráficos na execução de tarefas de propósito geral.

Referências

Khronos Group. (2013) "OpenCL - The open standard for parallel programming of heterogeneous systems", http://www.khronos.org/opencl, Agosto.

Manavski, S. A. e Valle, G. (2013) "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment", http://www.biomedcentral.com/1471-2105/9/S2/S10, Julho.

Matloff, N. (2013) "Programming on Parallel Machines", http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf, Agosto.

MPI. (2012) "MPI: A Message-Passing Interface Standard Version 3.0", Message Passing Interface Forum.

NVIDIA. (2013) "Plataforma de Computação Paralela", http://www.nvidia.com.br/object/cuda_home_new_br.html, Julho.

OpenMP. (2013) "OpenMP Application Program Interface – Version 4.0", http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, Agosto.

Ribeiro, N. S. e Fernandes, L. G. L. (2013) "Programação Híbrida: MPI e OpenMP", http://www.inf.pucrs.br/gmap/pdfs/Neumar/Artigo-IP2%20-%20Neumar%20Ribeiro.pdf, Maio.

Santos, T. S. e Lima. K. A. B. e Costa, D. A. da. e Aires, K. R. T. (2013) "Algoritmos de Visão Computacional em Tempo Real Utilizando CUDA e OpenCL", http://www.die.ufpi.br/ercemapi2011/artigos/ST4_20.pdf, Agosto.

Sena, M. C. R. e Costa, J. A. C. (2008) "Tutorial OpenMP C/C++", Maceio, Programa Campus Ambassador HPC - SUN Microsystems.

Silva, L. N. (2006) "Modelo Híbrido de Programação Paralela para uma Aplicação de Elasticidade Linear Baseada no Método dos Elementos Finitos", Brasília, UnB.