Otimizando a correspondência de *patches* para o *inpainting* de imagens com diferentes interfaces de programação paralela*

Luan Pereira[1], Leonardo Castro[1], Matheus S. Serpa[4], Adriano Q. de Oliveira[2], Fábio D. Rossi[3], Marcelo C. Luizelli[1], Philippe O. A. Navaux[4], Antonio Carlos S. Beck[4] e Arthur F. Lorenzon[1]

[1]Universidade Federal do Pampa – Campus Alegrete – RS – Brasil

[2]Universidade Federal de Santa Maria – Campus Cachoeira do Sul – RS – Brasil

[3]Instituto Federal Farroupilha – Campus Alegrete – RS – Brasil

[4]Universidade Federal do Rio Grande do Sul – Porto Alegre – RS – Brasil

luanvargas.aluno@unipampa.edu.br

Abstract. *Many problems in the image processing area require a high computational effort, e.g., inpainting methods based on the replication of patches. These methods enable the solution of real problems, such as the reconstruction of regions without content in images. Therefore, they can benefit from thread-level parallelism exploitation through the use of different parallel programming interfaces (PPIs). However, as each PPI has different characteristics w.r.t. the thread management, choosing the right one to implement a given application is important to achieve the best tradeoff between performance and energy consumption, represented by the energy-delay product (EDP). Given that, we analyze the potential of TLP exploitation of a well-known inpainting algorithm with different PPIs (PThreads, OpenMP, OmpSs-2, and OpenACC) and show which one provides the best performance, energy consumption, and EDP for three multicore architectures and two GPUs. When running a different set of experiments, we show that OpenMP exploiting TLP through parallel loops is better for AMD processors while OmpSs-2 is better for Intel processors.*

Resumo. *Diversos problemas da área de processamento de imagens demandam um alto esforço computacional, como, por exemplo, os métodos de inpainting baseados na replicação de patches. Estes métodos viabilizam a solução de problemas reais, como a reconstrução de regiões sem conteúdo em imagens. Portanto, eles podem se beneficiar da exploração do paralelismo no nível de threads (TLP) através de interfaces de programação paralela (IPPs). No entanto, como cada IPP possui diferentes características com respeito ao gerenciamento de threads, escolher a ideal para implementar uma aplicação é importante para obter o melhor custo-benefício entre desempenho e consumo de energia, representado pelo energy-delay product (EDP). Considerando o exposto, neste trabalho, nós analisamos o potencial de exploração de paralelismo de um algoritmo de inpainting amplamente difundido na literatura com diferentes IPPs (PThreads, OpenMP, OmpSs-2 e OpenACC) e mostramos qual IPP proporciona o melhor desempenho, consumo de energia e EDP para três arquiteturas multicore e duas GPUs. Através de um conjunto de experimentos, os resultados mostram que OpenMP explorando TLP com laços paralelos é melhor para processadores AMD, enquanto que o OmpSs-2 apresenta melhores resultados nos processadores Intel.*

*Este trabalho foi parcialmente financiado pela FAPERGS nos projetos 19/2551-0001224-1 e 19/2551-0001689-1 e PIBIC-CNPq

1. Introdução

A exploração do paralelismo no nível de *threads* (TLP – *thread-level parallelism*), onde múltiplos processadores simultaneamente executam partes do mesmo programa, vem sendo amplamente utilizada para acelerar a execução de aplicações da área de processamento de imagens. Para facilitar o desenvolvimento de tais aplicações e tornar o processo mais transparente para o programador, interfaces de programação paralela (IPPs) são usadas, como, por exemplo, PThreads (Posix Threads), OpenMP (*Open Multi-Processing*), OmpSs-2 e OpenACC. No entanto, cada IPP tem diferentes características com relação ao gerenciamento de *threads*, distribuição da carga de trabalho, comunicação e sincronização, que afetam o comportamento da aplicação [Lorenzon et al. 2017, Lorenzon et al. 2015, dos Santos Marques et al. 2017]. Portanto, escolher a IPP ideal para paralelizar uma aplicação é importante para obter o melhor desempenho, consumo de energia e o custo benefício entre ambos, representado pelo *energy-delay product* (EDP).

Uma grande variedade de problemas da área de processamento de imagens podem se beneficiar desta escolha, como síntese de textura, preenchimento de imagens, restauração de pinturas/fotografias danificadas, remoção/substituição de objetos selecionados em imagens, dentre outros. Embora tenham propósitos diferentes, estes problemas podem compartilhar a mesma solução, i.e., algoritmos de *inpainting* baseados em *patches* (a.k.a. exemplares). Dentre os diversos algoritmos publicados na literatura, destaca-se o proposto por Criminisi, Perez e Toyama [Criminisi et al. 2004], por viabilizar a solução de todos estes problemas (veja a Figura 1). Neste algoritmo, são realizadas repetidas buscas exaustivas em uma imagem base, comparando-a com um *patch* alvo, para identificar a região que apresenta maior similaridade com o exemplar. Este processo torna possível a replicação de conteúdo, tanto para a reconstrução de regiões vazias em imagens quanto para a reprodução de padrões de textura.

Considerando que este algoritmo de *inpainting* é amplamente utilizado em diversas aplicações da vida real e que o seu tempo de execução é extremamente alto para imagens de alta qualidade, as contribuições deste trabalho são as seguintes: (*i*) análise do potencial de exploração de paralelismo do algoritmo; (*ii*) implementações paralelas com diferentes IPPs: *PThreads*, *OpenMP parallel for*, *OpenMP tasks*, *OmpSs-2* e *OpenACC*; (*iii*) análise de desempenho, consumo de energia e EDP das implementações paralelas; e (*iv*) identificar a melhor IPP para cada plataforma *multicore* dada a métrica definida pelo usuário (desempenho, energia ou EDP). Através da execução de cada implementação paralela em diferentes configurações (tamanho de imagem e grau de exploração de paralelismo) em três arquiteturas *multicore* e duas GPUs, foi identificado que:

- Para processadores AMD, a exploração do paralelismo com OpenMP *parallel for*

Figura 1. Remoção de objeto de imagem (jogador de branco), cuja região alvo foi preenchida pelo algoritmo de *inpainting* de [Criminisi et al. 2004]

apresenta melhores desempenho, energia e EDP. Considerando o EDP, ele é 48% melhor que OmpSs-2, 8.4% que OpenMP *tasks* e 51% melhor que PThreads.
- Para processadores Intel, OmpSs-2 apresentou melhores resultados. Considerando o caso mais significativo, o EDP foi 61% melhor que OpenMP *parallel for*, 60% que OpenMP *tasks* e 78% melhor que PThreads.
- Para as arquiteturas GPUs, a implementação paralela com OpenACC não apresentou melhor desempenho quando comparado a execução paralela na arquitetura *host* devido a alta dependência de dados presente na aplicação que implica na quantidade de troca de dados entre a memória do dispositivo (GPU) e *host*.

O restante do artigo está organizado da seguinte maneira. Na Seção 2, discutem-se os trabalhos relacionados e são destacadas as contribuições deste artigo. A fundamentação teórica está descrita na Seção 3. A estratégia adotada na paralelização com cada IPP é apresentada na Seção 4. A metodologia utilizada é descrita na Seção 5 enquanto que os resultados são discutidos na Seção 6. Por fim, a conclusão é apresentada na Seção 7.

2. Trabalhos Relacionados

a) Otimização de Algoritmos de Processamento de Imagens. [Zhang 2009] apresentam P-SURF, uma abordagem paralela via PThreads do algoritmo de recuperação de imagem *SURF* (Speeded-Up Robust Features). Na mesma linha, [Fang et al. 2011] analisam as características de paralelismo do *SURF* e propõem uma implementação paralela com o *OpenSURF*, uma biblioteca feita em *OpenCV*. [Park et al. 2010] avaliam diferentes algoritmos de processamento de imagem (*Multiview Stereo Matching, Linear Feature Extraction, JPEG2000 Enconding (DWT), JPEG2000 Enconding (Tier-1), Cartoon-Style NPR e Oily-Style NPR* implementados em CUDA e OpenMP. [Kim et al. 2014] propõem uma implementação paralela do algoritmo de convolução 2D. O artigo tem como objetivo explorar a eficiência das instruções SIMD (*Single Instruction Multiple Data*) e da biblioteca TBB (*Threading Building Block*) nos processadores Intel. [Pratama and Ratno 2020] propõe uma implementação paralela de uma aplicação utilizada na captura de filmagens de auto estrada. [AlZu'bi et al. 2020] apresentam uma implementação paralela em GPU do algoritmo de segmentação de imagens médicas *Fuzzy C-Means* (FCM).

b) Comparação de Interfaces de Programação Paralela. [Balladini et al. 2011] analisam a influência da exploração do paralelismo com OpenMP e MPI no consumo de energia de aplicações paralelas. [Porterfield et al. 2013] apresentam uma análise dos fatores que afetam o desempenho e consumo de energia de aplicações implementadas com OpenMP. [Lorenzon et al. 2015] e [Lorenzon et al. 2015] apresentam uma avaliação da influencia de IPPs (PThreads, OpenMP e MPI) na eficiência energética de diversas aplicações paralelas com diferentes níveis de comunicação para processadores embarcados. [Butko et al. 2017] avaliam o custo beneficio entre desempenho e consumo de energia de dois modelos de programação (OpenMP 3.0 e OmpSs) em arquiteturas heterogêneas. [Castello et al. 2020] demonstra os benefícios de usabilidade e desempenho das soluções *Lightweight thread*.

c) Nossas Contribuições. Conforme discutido anteriormente, a implementação de algoritmos paralelos na área de processamento de imagens não contempla o método de *inpainting* [Criminisi et al. 2004]. Por outro lado, os trabalhos que avaliam a melhor IPP para paralelizar um determinado tipo de aplicação não contemplam o algoritmo de *inpainting* utilizado em vários problemas da vida real. Portanto, este é o primeiro trabalho

que (*i*) analisa e propõe diferentes implementações paralelas para otimizar a execução do algoritmo de *inpainting* e (*ii*) identifica qual a melhor estratégia de paralelização para cada plataforma *multicore* de acordo com a métrica definida pelo usuário.

3. Fundamentação Teórica

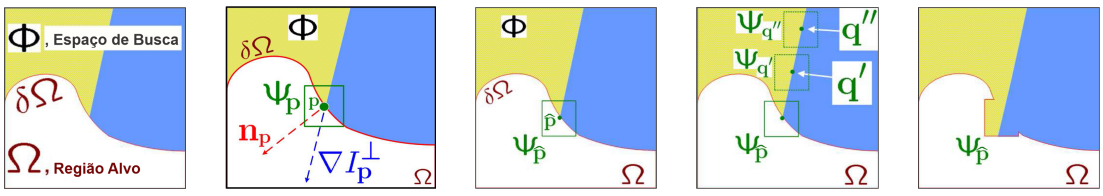
3.1. Algoritmo de *Inpainting*

O Algoritmo 1. detalha o método de *inpainting*. A entrada do algoritmo compreende a imagem alvo $\mathcal{I}$ e uma imagem binária Ω com a região a ser reconstruída indicada. Com base nisso, *patches* (exemplares quadrados) são selecionados em uma região de busca $\Phi \in \mathcal{I}$ por similaridade de cor e copiados para o interior de Ω. A região de busca pode ser delimitada automaticamente com $\Phi = \mathcal{I} - \Omega$, como mostrado na Figura 2(a). Para produzir uma reconstrução adequada, o algoritmo de *inpainting* repete três processos iterativamente: (*i*) definição do *patch* com maior prioridade de preenchimento (linhas 3 a 5 do Algoritmo 1.); (*ii*) busca pelo exemplar mais adequado ao preenchimento dos *pixels* vazios do *patch* alvo em Φ (linha 6 do Algoritmo 1.); e (*iii*) atualização dos valores de confiança nos *pixels* preenchidos (linha 8 do Algoritmo 1.). O algoritmo encerra sua execução quando todos os *pixels* indicados inicialmente em Ω estiverem preenchidos.

3.1.1. Prioridade de Preenchimento

Para reconstruir adequadamente a região alvo Ω, uma função que mensura a prioridade de preenchimento de cada *patch* $\Psi_{\boldsymbol{p}}$ centrado em cada ponto $\boldsymbol{p}$ da borda $\delta\Omega$ de Ω é utilizada. Ela prioriza *patches* que possuem simultaneamente a maior probabilidade de estar em uma região de continuação de borda e com informação confiável em seu entorno. Esta estratégia faz com que, ao longo da reconstrução, a estrutura da cena seja preservada e que a informação mais confiável sirva como base para a busca pelo melhor candidato ao preenchimento. A prioridade $P(\boldsymbol{p})$ para um dado *patch* $\Psi_{\boldsymbol{p}}$, centrado em um ponto $\boldsymbol{p} \in \delta\Omega$, se dá pela equação $P(\boldsymbol{p}) = C(\boldsymbol{p})D(\boldsymbol{p})$, onde $C(\boldsymbol{p})$ e $D(\boldsymbol{p})$ correspondem aos termos de confiança e dados, respectivamente.

O termo $C(\boldsymbol{p})$ permite medir a quantidade de informação confiável em cada *patch* $\Psi_{\boldsymbol{p}}$ candidato ao preenchimento. Neste termo, quanto mais próximo do conteúdo original



(a) (b) (c) (d) (e)

Figura 2. Passo a passo para a propagação de estrutura no processo de síntese textura baseado em *patches*: (a) imagem alvo $\mathcal{I}$, com a região alvo Ω e sua borda $\delta\Omega$ destacadas; (b) ilustração dos vetores relacionados ao cômputo de $D(p)$; (c) *patch* $\Psi_{\hat{p}}$ que maximiza $P(\boldsymbol{p})$; (d) exibição de candidatos ($\Psi_{q'}$ e $\Psi_{q''}$) ao preenchimento de $\Psi_{\hat{p}}$; (e) resultado do preenchimento dos *pixels* vazios de $\Psi_{\hat{p}}$ com o conteúdo do candidato selecionado $\Psi_{\hat{q}}$

Algoritmo 1. Pseudocódigo do Método de *Inpainting* em Modo Sequencial

Input: $\mathcal{I}$ imagem alvo, Ω máscara (imagem binária)
Output: $\mathcal{I}$ preenchida
1: Inicializar variáveis
2: **while** Ω ! = $\emptyset$ **do**
3: Extração da borda $\delta\Omega$ de Ω
4: Cômputo das prioridades $P(\boldsymbol{p})$ $\forall \boldsymbol{p} \in \delta\Omega$
5: Encontrar o *patch* $\Psi_{\widehat{\boldsymbol{p}}}$ com máxima prioridade, i.e., $\widehat{\boldsymbol{p}} = \arg\max_{\boldsymbol{p}\in\delta\Omega} P(\boldsymbol{p})$
6: Buscar pelo *patch* $\Psi_{\widehat{\boldsymbol{q}}} \in \Phi$ que minimiza $d(\Psi_{\widehat{\boldsymbol{p}}}, \Psi_{\boldsymbol{q}})$
7: Copiar o conteúdo de $\Psi_{\widehat{\boldsymbol{q}}}$ para $\Psi_{\widehat{\boldsymbol{p}}}$ $\forall \boldsymbol{p} \in \Psi_{\widehat{\boldsymbol{p}}} \cap \Omega$
8: Atualizar $C(\boldsymbol{p})$ $\forall \boldsymbol{p} \in \Psi_{\widehat{\boldsymbol{p}}} \cap \Omega$
9: **end while**
10: Retornar $\mathcal{I}$ após o preenchimento da área delimitada em Ω

da imagem o *patch* estiver, maior será o seu valor de confiança. Já o termo $D(\boldsymbol{p})$ tem como objetivo forçar que a propagação de estruturas lineares (ditos *isophotes*) incidentes em $\delta\Omega$ seja realizada prioritariamente. A Figura 2(b) ilustra a relação dada pelo vetor unitário normal a borda $\boldsymbol{n}_p$ e pelo vetor $\nabla I_p^{\perp}$ que representa o *isophote* em termos de direção e intensidade no ponto $\boldsymbol{p}$. Neste caso, quanto menor for o ângulo entre os dois vetores, maior será a prioridade. A função de prioridade é computada para cada *patch* da borda. Após, seleciona-se para o preenchimento o *patch* $\Psi_{\widehat{\boldsymbol{p}}}$ que maximiza $P(\boldsymbol{p})$. A Figura 2(c) exibe o *patch* selecionado em uma iteração de exemplo. Como pode ser observado, $\Psi_{\widehat{\boldsymbol{p}}}$ está centrado no ponto $\widehat{\boldsymbol{p}}$ que contém um *isophote* em destaque, e que também possui informação confiável em seu entorno, uma vez que a prioridade do *patch* é dada pelo produto dos dois termos.

3.1.2. Buscando o Melhor *Patch* para o Preenchimento

Após selecionar $\Psi_{\widehat{\boldsymbol{p}}}$, faz-se uma busca exaustiva em Φ, para identificar o *patch* mais similar $\Psi_{\widehat{\boldsymbol{q}}}$ para o seu preenchimento. Mais precisamente, computa-se:

$$\Psi_{\widehat{q}} = \arg\min_{\Psi_q \in \Phi} d(\Psi_{\widehat{p}}, \Psi_q), \tag{1}$$

onde $d(\Psi_{\widehat{p}}, \Psi_q)$ corresponde a soma das diferenças quadradas no espaço de cores CIE Lab entre quaisquer dois *patches* genéricos. No exemplo da Figura 2(d), dois dos *patches* mais similares a $\Psi_{\widehat{\boldsymbol{p}}}$ encontrados na busca exaustiva estão destacados ($\Psi_{\boldsymbol{q}'}$ e $\Psi_{\boldsymbol{q}''}$). Visualmente, pode-se perceber que estes são similares – no comparativo *pixel* a *pixel* – a região não vazia de $\Psi_{\widehat{\boldsymbol{p}}}$. Dentre estas e outras possibilidades, seleciona-se o *patch* que minimiza $d(\Psi_{\widehat{p}}, \Psi_q)$ para o preenchimento, produzindo o resultado exibido na Figura 2(e).

3.1.3. Atualizando os Termos de Prioridade

Na última etapa, são definidos os valores de confiança para os *pixels* preenchidos na iteração corrente, com o valor de confiança de $C(\widehat{\boldsymbol{p}})$. Esta estratégia de atualização faz com que os valores do termo em Ω decaiam gradativamente, produzindo um efeito concêntrico de redução da confiabilidade.

3.2. Interfaces de Programação Paralela

OpenMP consiste de um conjunto de diretivas de compilação, funções de biblioteca e variáveis de ambiente. O paralelismo é explorado através da inserção de diretivas no código sequencial que informam ao compilador como e quais partes do código deverão ser executadas em paralelo. Foram utilizados dois modelos de programação paralela: (*i*) *parallel for*, onde as iterações de um laço *for* são divididas entre as *threads*; e (*ii*) com *tasks*, que é uma entidade de execução mínima que pode ser gerenciada independentemente pelo escalonador do sistema [OpenMP Arch. Review Board 2018]. ***PThreads*** é uma IPP onde suas funções permitem ajuste fino no tamanho do grão da carga de trabalho. Assim, a exploração do paralelismo através do gerenciamento das *threads* (e.g., criação/finalização das *threads*, distribuição da carga de trabalho e controle de execução) é completamente definido pelo programador [Butenhof 1997].

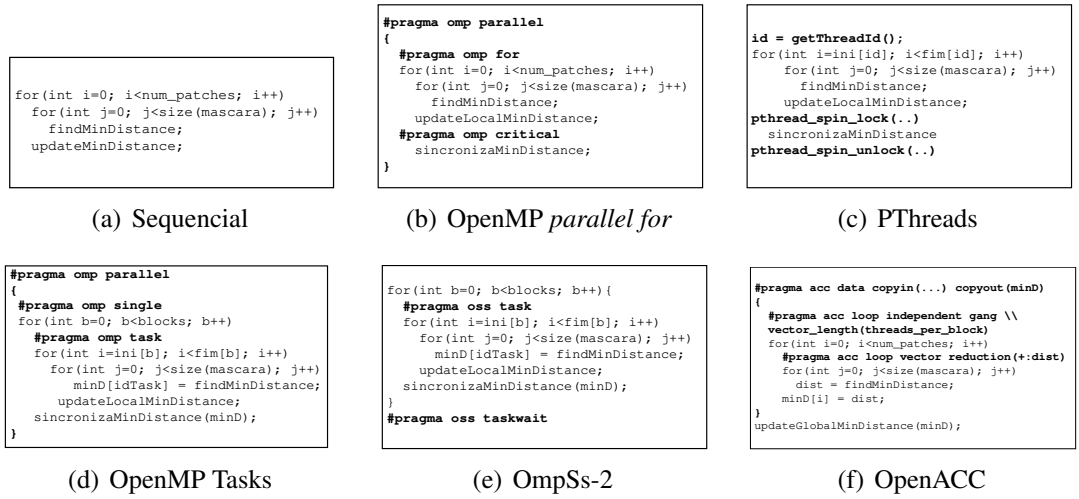
OmpSs-2 consiste de um conjunto de diretivas e funções de biblioteca. O paralelismo é explorado no nível de *tasks*, onde o usuário deve inserir notações de código para identificar como o código paralelo será gerado. Diferentemente do OpenMP *tasks*, uma aplicação *OmpSs-2* inicia a execução com todas as *threads*. Nele, a *thread* principal executa as regiões sequenciais e é responsável por criar as *tasks* (identificadas no código através de diretivas) que serão atribuídas às threads que ficam aguardando por carga de trabalho. O *OmpSs-2* implementa também um gerenciador que é responsável por definir em tempo de execução qual *thread* executará cada *task*, o número de *threads*, a alocação de *threads*, entre outras funcionalidades [Barcelona Supercomputing Center 2020]. ***OpenACC*** também é composto de diretivas de compilação, funções de biblioteca e variáveis de ambiente que auxiliam o desenvolvimento de aplicações paralelas. Neste sentido, as diretivas identificam quais regiões de código deverão ser executadas em paralelo no dispositivo acelerador. O programador deve também identificar quais dados devem ser copiados do *host* para o dispositivo e vice-versa [OpenACC Working Group and others 2011].

4. Implementação Paralela do Algoritmo de *Inpainting*

A Tabela 1 apresenta o tempo de execução de cada etapa do algoritmo descrita na Seção 3.1: Inicialização do algoritmo; Etapa 1 (prioridade de preenchimento); Etapa 2 (buscando o melhor *patch* para o preenchimento); e Etapa 3 (atualização dos termos de prioridade). Os dados foram retirados da execução do algoritmo sequencial sobre duas imagens de entrada com diferentes tamanhos nas três arquiteturas multicore utilizadas neste trabalho (Tabela 2). Conforme pode ser observado, a Etapa 2 é responsável por aproximadamente 90% do tempo total de execução, sendo a principal candidata para a exploração do paralelismo. Por outro lado, as Etapas 1 e 3 possuem tempo de computação no qual a exploração do paralelismo poderia levar a um maior *overhead* (e.g., criação de *threads*, distribuição da carga de trabalho e sincronização) para a aplicação. Por isso, optou-se por realizar a paralelização da Etapa 2, cujo pseudocódigo está descrito na Figura 3(a).

Tabela 1. Proporção de tempo de execução de cada etapa do algoritmo

	AMD Ryzen 9 3900		**Intel Xeon E5-2650 v3**		**Intel Xeon E5-2699 v4**	
	300×300	1280×720	300×300	1280×720	300×300	1280×720
Inicialização	1.17s	130.89s	1.69s	191.4s	1.41s	146.244s
Etapa 1	0.01s	1.09s	0.02s	1.47s	0.02s	0.611s
Etapa 2	9.46s	1019.88s	14.45s	1857.8s	11.69s	1530.62s
Etapa 3	0.0002s	0.018s	0.0003s	0.01s	0.0004s	0.0049s
Total	10.65s	1151.88s	16.17s	2050.74s	13.13s	1677.48s



(a) Sequencial (b) OpenMP *parallel for* (c) PThreads

(d) OpenMP Tasks (e) OmpSs-2 (f) OpenACC

Figura 3. Implementações paralelas do algoritmo de *impainting*

O pseudocódigo das implementações paralelas são apresentados na Figura 3. De uma maneira geral, foi utilizada a estratégia de decomposição da imagem (representada matricialmente 2-D) por blocos de linhas, sendo paralelizado o laço externo que percorre a imagem (e.g., matriz). O número de blocos é definido pelo usuário, que pode ser igual ao número de *threads* que está executando a aplicação. Ao final da computação paralela de cada *thread*, cada uma terá um valor privado contendo as informações do melhor *patch* para o preenchimento. Portanto, uma região síncrona foi inserida para realizar a sincronização e obter as informações globais do melhor *patch* para o preenchimento.

Para a implementação com OpenMP *parallel for* (Figura 3(b)), foram utilizadas as diretivas *#pragma omp parallel* para criação das *threads* e *omp for* para distribuir a carga de trabalho entre as *threads*. Foi utilizado o *schedule* padrão do OpenMP. Por fim, após cada *thread* obter as informações do melhor *patch* (*findMinDistance*), a diretiva *#pragma omp critical* garante a sincronização destes valores. Como nesta região o importante é o endereço (posição na imagem em linha-coluna) do melhor *patch* e não o valor encontrado pelo *findMinDistance*, a clausula *reduction* não pode ser utilizada. Na implementação com PThreads, foi desenvolvida uma função auxiliar para calcular a carga de trabalho de cada *thread*. Portanto, cada *thread* irá computar apenas as iterações definidas para o seu *id*. Por fim, para a sincronização, foi utilizada a função *pthread_spin_lock*, uma vez que nos experimentos preliminares ela apresentou melhor desempenho que o *pthread_mutex*.

Para as implementações com *tasks*, a seguinte estratégia foi utilizada. As linhas da imagem foram divididas em blocos (de acordo com o número definido pelo usuário), e para cada bloco uma *task* é criada. Para o OpenMP *tasks* (Figura 3(d)), uma nova task é criada pela diretiva *#pragma omp task* enquanto que para o OmpSs-2, a diretiva *#pragma oss task* define a criação de uma nova *task* (Figura 3(e)). Durante a execução paralela, cada *task* atualiza seu valor de *minDistance* em um vetor compartilhado entre as *threads* que será utilizado pela *thread* principal realizar a sincronização do valor global de *minD*. Por fim, na versão do OmpSs-2 é inserido a diretiva *#pragma oss taskwait* para indicar uma barreira na execução do programa. Isto é, a *thread* principal só pode continuar a execução quando todas as *tasks* criadas tiverem finalizadas.

Por fim, para a implementação em *OpenACC*, o construtor ***data*** é usado para defi-

nir as variáveis que serão alocadas na memória do dispositivo (GPU) durante a execução da região paralela e quais dados devem ser copiados do *host* para dispositivo e vice-versa. Neste caso, a imagem (e.g., matriz) e a mascara são copiadas para o dispositivo toda a vez que a região paralela for executada e houver mudanças nos dados, enquanto que o vetor resultante *minD*, contendo o melhor *patch* de cada *thread* é copiado para o dispositivo, onde a sincronização é realizada. Então as diretivas *acc loop* são usadas para dividir as iterações dos laços *for*, utilizando a redução na variável *dist*.

5. Metodologia

Os experimentos foram realizados em três arquiteturas multicores e duas GPUs, conforme mostrado na Tabela 2. Foi utilizado o Sistema Operacional Linux Debian 10. A frequência de cada CPU foi configurada para ajustar de acordo com a carga de trabalho, através do *governor* DVFS *ondemand*, que é usado na maioria das versões Linux. Cada aplicação executada nas arquiteturas multicore foi compilada com GCC/G++ 9.2, usando a flag de otimização $-O3$. Já as aplicações implementadas com OpenACC foram compiladas com *pgcc* 19.10. Para a implementação *OmpSs-2*, utilizou-se também o compilador Mercurium, uma infraestrutura *source-to-source* para a geração do código paralelo.

O tempo de execução de cada aplicação foi obtido pela função *get_time_of_day()*. Por outro lado, o consumo de energia foi obtido diretamente dos contadores de hardware presentes nos processadores modernos. No caso do processador Intel Xeon, a biblioteca *Running Average Power Limit* (RAPL) foi usada [Hähnel et al. 2012], enquanto que a biblioteca *Application Power Management* foi usada para o processador AMD Ryzen [Hackenberg et al. 2013]. Já o consumo energia de todo o sistema executando as aplicações na GPU foi obtido através do IPMI (*Intelligent Platform Management Interface*) [Slaight 2002]. O EDP da execução de cada aplicação foi obtido pela multiplicação do tempo de execução pelo consumo de energia. Para avaliar a escalabilidade de cada IPP conforme o conjunto de entrada muda, foram utilizadas duas imagens com diferentes tamanhos (x e y): 300×300 e 1280×720. O tamanho de máscara associado a cada imagem, respectivamente, foi: *4040* e *59292* pixel. Cada aplicação paralela foi executada com diferentes números de *threads* (Intel e AMD) e *threads* por bloco (GPU). Durante os experimentos, foi utilizada a política de afinidade *compact*, que mantém threads vizinhas em núcleos próximos. Os resultados apresentados a seguir são uma média de dez execuções com um desvio padrão menor que 0.7%.

6. Resultados

6.1. Análise de Desempenho, Consumo de Energia e EDP

A Figura 4 apresenta os resultados de desempenho de cada implementação paralela para a execução em cada arquitetura *multicore* e tamanho de imagem. O desempenho é mostrado

Tabela 2. Características das arquiteturas *multicore*

	AMD Ryzen 9 3900	**Intel 2x Xeon E5-2650 v3**	**Intel 2x Xeon E5-2699 v4**
Microarquitetura	Zen+	Haswell	Broadwell
Número de núcleos	12	10	22
Número de threads	24	40	88
Cache L3 (total)	64MB	50MB	55MB
Memória RAM	32GB	128GB	256GB
GPU	–	NVIDIA Tesla K80, Kepler, 2496 CUDA cores	NVIDIA Tesla P100, Pascal, 3584 CUDA cores

através do *speedup*, isto é, a aceleração proporcionada pela execução paralela quando comparada à execução com uma única *thread*. Para as execuções na GPU, o *baseline* é o resultado da versão sequencial no *host*. Para o processador AMD, o melhor desempenho de cada IPP foi obtido na execução com 24 *threads* independente do tamanho da imagem. Na média das duas imagens, OpenMP *parallel for* (*omp_for*) foi 27% mais rápido que *OmpSs-2*, 5.2% que *omp-task* e 43% mais rápido que *PThreads*. Já nos processadores Intel, a IPP que fornece o melhor desempenho varia de acordo com o tamanho da imagem. Para a menor imagem avaliada (300×300), *omp-for* apresentou melhor desempenho enquanto que para imagens maiores, *OmpSs-2* foi mais rápido: 35% melhor que *omp-for*, 33.8% melhor que *omp-task* e 53.4% melhor que *PThreads*. Por fim, as execuções da implementação com OpenACC nas duas GPUs alvo produziu pior desempenho do que o observado nas arquiteturas de propósito geral.

As implementações com OpenMP (*omp-for* e *omp-task*) possuem melhor escalabilidade na arquitetura *multicore* com acesso uniforme à memória (AMD 24Core) do que quando comparado ao comportamento nas máquinas com acesso não uniforme à memória (NUMA - *non-uniform memory access*, Intel 40Core e 88Core). Este comportamento justifica a falta de escalabilidade nos processadores Intel. Por outro lado, de uma maneira geral, a implementação com *OmpSs-2* atinge melhor desempenho em arquiteturas NUMA devido a maneira como o gerenciamento de *tasks* é realizado em tempo de execução: (*i*) as *threads* são criadas apenas uma vez, no começo da execução, e portanto, não existe *overhead* de criação de *threads* sempre que uma região paralela é encontrada; (*i*) a distribuição da carga de trabalho se dá de maneira que os recursos computacionais não fiquem sub ou super utilizados; e (*iii*) o gerenciamento inteligente da alocação de recursos identifica o número de núcleos ideais para executar a aplicação, de acordo com o uso dos recursos computacionais. Este comportamento não ocorre no processador AMD pois o espaço de exploração em que o sistema de gerenciamento em tempo de execução do *OmpSs-2* pode atuar é limitado a apenas um *socket*. Já a implementação com *PThreads* apresentou os piores resultados de desempenho para os processadores *multicore* devido ao alto custo computacional da (*i*) criação de *threads* a cada iteração do algoritmo; e (*ii*) da sincronização para atualização dos valores calculados em cada *thread* através do *spin_lock*. Portanto, a possibilidade de explorar melhor a distribuição da carga de trabalho em *PThreads* não é suficiente para reduzir o *overhead* de criação de *threads* e sincronização de dados [Lorenzon and Beck 2019].

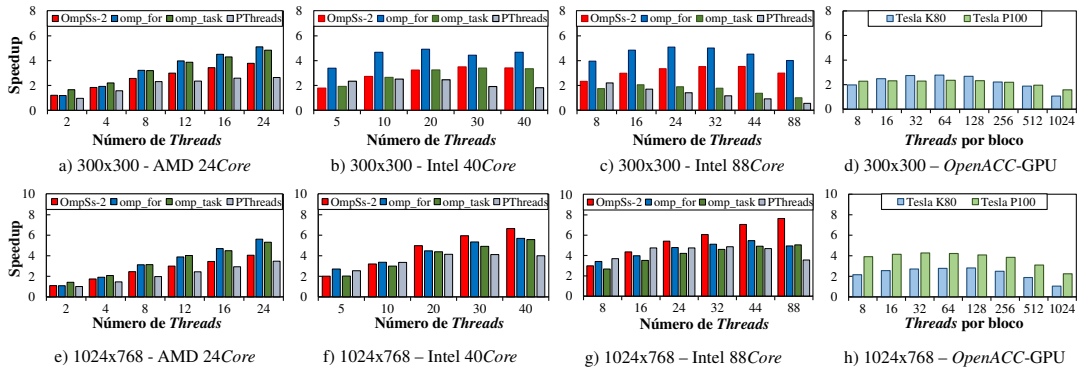


Figura 4. Desempenho (*speedup* sobre a versão sequencial) de cada IPP em cada arquitetura (quanto maior a barra, melhor)

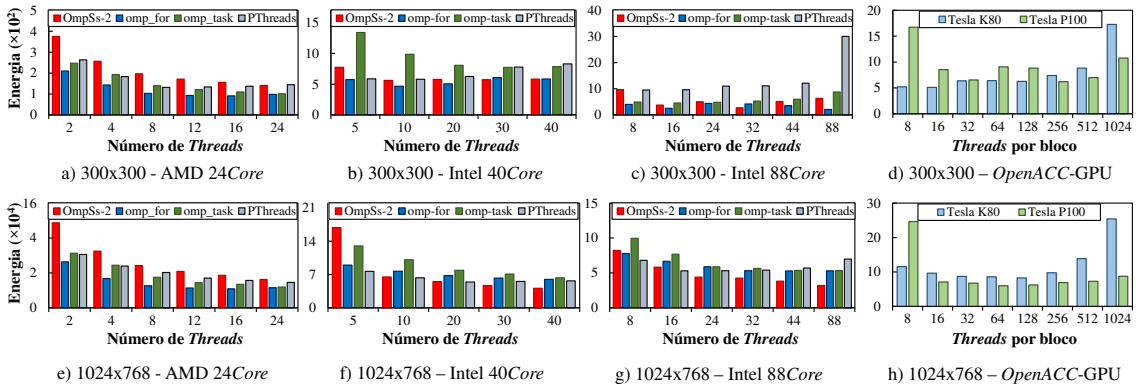


Figura 5. Consumo de energia (quanto menor, melhor)

Dada a natureza de dependência de dados entre cada etapa do algoritmo de *inpainting* (conforme discutido na Seção 3.1), existe a necessidade de sincronização e comunicação no início e fim de cada região paralela. Portanto, quando a aplicação é executada na GPU com *OpenACC*, para cada iteração, há a necessidade de transferir os dados do *host* para a GPU e vice-versa. Esta transferência de dados é custosa e responsável por aproximadamente 90% do tempo total de execução da aplicação (valores obtidos através do *profile* da aplicação com o NVidia *Visual Profiler* (*nvprof*). Neste sentido, a implementação deste método de *inpainting* em arquiteturas GPU não é capaz de produzir melhor desempenho que em processadores de propósito geral.

O comportamento de cada IPP discutido acima, com relação ao uso dos recursos computacionais e desempenho, afeta diretamente o consumo de energia e o EDP. As Figuras 5 e 6 apresentam os resultados de consumo de energia e EDP, em dados brutos, para cada configuração (IPP, número de *threads*, imagem de entrada e arquitetura). Portanto, quanto menor for o valor, melhor o resultado. O comportamento observado foi muito similar ao desempenho: para o processador AMD, a implementação *omp-for* apresentou melhores resultados de energia e EDP, enquanto que nos processadores Intel, a implementação *OmpSs-2* foi melhor para a imagem de alta definição. No melhor caso de EDP do processador AMD para a imagem de alta definição, *omp-for* foi 48% melhor que *OmpSs-2*, 8.4% que *omp-task* e 51% melhor que *PThreads*. Já no processador Intel 88-Core, *OmpSs-2* foi 61% melhor que *omp-for*, 60% que *omp-task* e 78% melhor que *PThreads*. Por fim, a implementação com *OpenACC* não apresentou resultados satisfatórios de energia e EDP devido as razões discutidas anteriormente.

6.2. Otimizando o uso de IPPs

Com o objetivo de prover diretivas de qual IPP proporciona o melhor resultado para cada métrica em cada arquitetura, a Tabela 3 apresenta a melhor escolha de IPP para cada cenário. Conforme pode ser observado, as implementações OpenMP (*omp-for* e *omp-task*) apresentam os melhores resultados em duas situações: no processador AMD, independente do tamanho de imagem; e no processador Intel para a execução com uma imagem pequena. Adicionalmente, observa-se que o número de *threads* que proporciona o melhor resultado nos processadores Intel para a entrada pequena é significativamente menor do que o número de *threads* disponíveis no sistema, na maioria dos casos. Por outro lado, a IPP *OmpSs-2* apresenta os melhores resultados quando o tamanho da imagem é maior, nos processadores Intel, independente da métrica alvo.

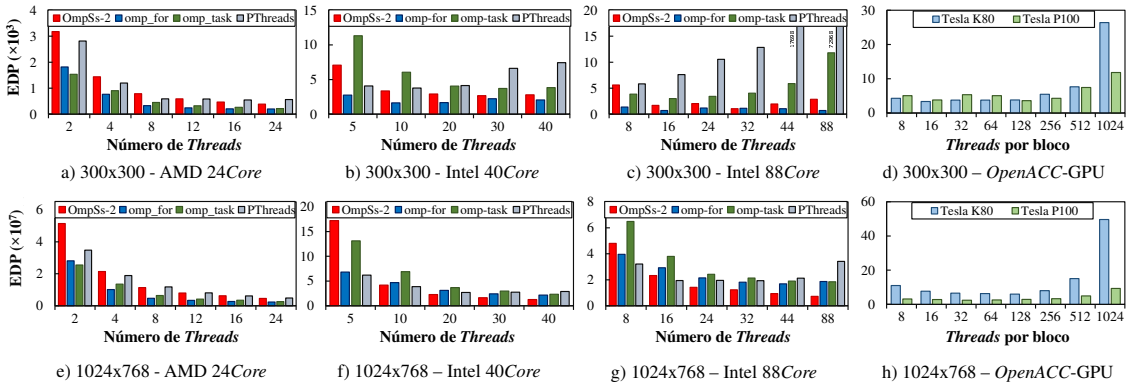


Figura 6. Resultados de EDP (quanto menor, melhor)

Tabela 3. Melhor resultado para cada arquitetura (IPP-Número de *Threads*)

	AMD Ryzen 9 3900			Intel Xeon E5-2650 v3 (Tesla K80)			Intel Xeon E5-2699 v4 (Tesla P100)		
	Desempenho	***Energia***	***EDP***	***Desempenho***	***Energia***	***EDP***	***Desempenho***	***Energia***	***EDP***
300x300	*omp_for*-24	*omp_for*-16	*omp_for*-24	*omp_for*-20	*omp_for*-10	*omp_for*-10	*omp_for*-24	*omp_for*-88	*omp_for*-16
1280x720	*omp_for*-24	*omp_for*-16	*omp_for*-24	*OmpSs-2*-40	*OmpSs-2*-40	*OmpSs-2*-40	*OmpSs-2*-88	*OmpSs-2*-88	*OmpSs-2*-88

7. Conclusão

Neste trabalho, nós analisamos o potencial de exploração de paralelismo de um algoritmo de *inpainting* altamente difundido na literatura e propôs implementações paralelas com diversas IPPs: PThreads, OpenMP, *OmpSs-2* e OpenACC. Através da execução em três arquiteturas *multicore* e duas GPUs, mostramos que OpenMP é melhor para processadores AMD enquanto que *OmpSs-2* é melhor para processadores Intel. Adicionalmente, dada a natureza de dependências do algoritmo de *inpainting*, mostrou-se que sua implementação paralela não é eficiente em arquiteturas GPUs. Como trabalhos futuros, pretendemos expandir o número de aplicações reais utilizadas na área de processamento de imagem.

Referências

AlZu'bi, S., Shehab, M., Al-Ayyoub, M., Jararweh, Y., and Gupta, B. (2020). Parallel implementation for 3d medical volume fuzzy segmentation. *Pattern Recognit. Lett.*, 130:312–318.

Balladini, J., Suppi, R., Rexachs, D., and Luque, E. (2011). Impact of parallel programming models and cpus clock frequency on energy consumption of hpc systems. In *IEEE/ACS AICCSA*, pages 16–21.

Barcelona Supercomputing Center (2020). OmpSs-2 Spec. Acesso em: 01/08/2020.

Butenhof, D. R. (1997). *Programming with POSIX Threads*. Addison-Wesley, USA.

Butko, A., Bruguier, F., Gamatié, A., and Sassatelli, G. (2017). Efficient programming for multicore processor heterogeneity: Openmp versus ompss. In *OpenSuCo*.

Castello, A., Mayo, R., Seo, S., Balaji, P., Quintana-Ortí, E. S., and Peña, A. J. (2020). Analysis of threading libraries for high performance computing. *IEEE Trans. Comp.*

Criminisi, A., Perez, P., and Toyama, K. (2004). Region filling and object removal by exemplar-based image inpainting. *IEEE Trans. Image Process.*, 13:1200–1212.

dos Santos Marques, W., de Souza, P. S. S., Lorenzon, A. F., Schneider Beck, A. C., Beck Rutzig, M., and Diniz Rossi, F. (2017). Improving edp in multi-core embedded systems through multidimensional frequency scaling. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4.

Fang, Z., Yang, D., Zhang, W., Chen, H., and Zang, B. (2011). A comprehensive analysis and parallelization of an image retrieval algorithm. In *ISPASS*, pages 154–164. IEEE.

Hackenberg, D., Ilsche, T., Schone, R., Molka, D., Schmidt, M., and Nagel, W. E. (2013). Power measurement techniques on standard compute nodes: A quantitative comparison. In *ISPASS*, pages 194–204. IEEE.

Hähnel, M., Döbel, B., Völp, M., and Härtig, H. (2012). Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval.*, 40:13–17.

Kim, C. G., Kim, J. G., et al. (2014). Optimizing image processing on multi-core cpus with intel parallel programming technologies. *Multimedia tools and Appl.*, 68:237–251.

Lorenzon, A. F. and Beck, A. C. S. (2019). *Parallel Computing Hits the Power Wall - Principles, Challenges, and a Survey of Solutions*. Springer Briefs in Computer Science. Springer.

Lorenzon, A. F., Dellagostin Souza, J., and Schneider Beck, A. C. (2017). Laant: A library to automatically optimize edp for openmp applications. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1229–1232.

Lorenzon, A. F., Sartor, A. L., Cera, M. C., and Beck, A. C. S. (2015). The influence of parallel programming interfaces on multicore embedded systems. In *IEEE COMPSAC*, volume 2, pages 617–625. IEEE.

Lorenzon, A. F., Sartor, A. L., Cera, M. C., and Schneider Beck, A. C. (2015). Optimized use of parallel programming interfaces in multithreaded embedded architectures. In *2015 IEEE Computer Society Annual Symposium on VLSI*, pages 410–415.

OpenACC Working Group and others (2011). The openacc application programming interface. *Retrieved March*, 26:2019.

OpenMP Arch. Review Board (2018). OpenMP API. V.5.0. Acesso em: 01/08/2020.

Park, I. K., Singhal, N., Lee, M. H., Cho, S., and Kim, C. (2010). Design and performance evaluation of image processing algorithms on gpus. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):91–104.

Porterfield, A. K., Olivier, S. L., Bhalachandra, S., and Prins, J. F. (2013). Power measurement and concurrency throttling for energy reduction in openmp programs. In *IPDPS, Workshops and Phd Forum*, pages 884–891. IEEE.

Pratama, Y. and Ratno, P. P. (2020). . *Indonesian J. of Comput. and Cybern. Syst.*, 14(3).

Slaight, T. (2002). Platform management ipmi controllers, sensors, and tools. In *IDF*.

Zhang, N. (2009). Computing parallel speeded-up robust features (p-surf) via posix threads. In *ICIC*, pages 287–296. Springer.