Paralelização de uma Aplicação de Detecção e Eliminação de Ruídos em Streaming de Vídeo com a DSL SPar

Renato B. Hoffmann, Dalvan Griebler, Luiz G. Fernandes

[1] Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Grupo de Modelagem de Aplicações Paralelas (GMAP), Porto Alegre – RS – Brasil

{renato.hoffmann,dalvan.griebler}@acad.pucrs.br

Resumo. *Restauração de imagem é uma importante etapa de qualquer sistema de computação gráfica. Este trabalho tem como objetivo apresentar e avaliar o paralelismo de Denoiser, uma aplicação para detecção e eliminação de ruído em streaming de vídeo. Foram avaliados o speed-up e programabilidade das interfaces SPar, Thread Building Blocks e FastFlow. Os resultados mostram que a SPar obteve bons resultados de programabilidade e desempenho.*

1. Introdução

Aplicações caracterizadas por um fluxo de dados vêm progressivamente crescendo em importância e difusão. De fato, aplicações de *streaming* como vídeo, áudio e imagem são recorrentes cargas de trabalho em computadores pessoais. Como são computacionalmente intensivas, essas aplicações tipicamente necessitam de paralelismo para obter resultados eficientemente. Nesse âmbito, a DSL (Linguagem Específica de Domínio) SPar [Griebler et al. 2017] foi proposta com o intuito de fornecer abstrações de paralelismo para o domínio de *stream*. Através de anotações no código fonte, a SPar propõe habilitar o paralelismo de *stream* sem a necessidade de refatorar o código fonte. Contudo, por ser uma interface emergente, ainda é necessário testar ela em aplicações reais.

Por outro lado, com a massiva produção de imagens e filmes digitais, muitas vezes gerados em más condições, filtros de restauração tornaram-se importantes etapas de sistemas de computação gráfica. Nesse contexto, destacamos Denoiser [Aldinucci et al. 2012], uma robusta aplicação para restauração de imagens corrompidas por ruído *Salt and Pepper*. Esse ruído se caracteriza por alterar o valor de variação do píxel para o máximo ou mínimo (0 ou 255 para uma imagem de 8 bits), gerando sobre a imagem um chuviscado preto e branco. Entretanto, por garantir alta qualidade de restauração, Denoiser possui uma vazão baixa, demandando paralelismo para fornecer resultados mais eficientemente.

Esse trabalho contribui com o desenvolvimento do paralelismo da aplicação Denoiser em duas novas interfaces: SPar e TBB [Reinders 2007]. Além disso, é fornecida uma análise comparativa de desempenho e programabilidade para as versões paralelas implementadas, incluindo uma própria versão com a interface FastFlow (seção 2.3 de [Aldinucci et al. 2014]). O artigo está estruturado da seguinte forma. Na Seção 2, são comparados e discutidos os trabalhos relacionados. A Seção 3 descreve a estrutura e estratégia de paralelismo da aplicação Denoiser. Subsequentemente, são apresentados e discutidos os resultados e experimentos na Seção 4. Por fim, a Seção 5 conclui o trabalho.

2. Trabalhos Relacionados

Os estudos de [Wan et al. 2010] abordam uma implementação paralela de um algoritmo de restauração de imagem em arquiteturas com memória distribuída. Em sua implementação, os autores utilizam um padrão mestre-escravo para processar diferentes segmentos de uma imagem em paralelo. Além disso, os autores fornecem uma avaliação do *speed-up* com até 6 processadores. Em comparação, nosso trabalho aborda o paralelismo de stream em arquiteturas com memória compartilhada e também apresenta uma avaliação de programabilidade. Outro fator de diferenciação é que nosso trabalho não prioriza a descrição das técnicas algorítmicas de restauração de imagem.

Em [Aldinucci et al. 2012], Denoiser é apresentado como uma aplicação para restauração de imagens corrompidas por *Salt and Pepper*. Os pesquisadores apresentam uma detalhada explicação algorítmica e o estado da arte das técnicas de restauração de imagem. Além disso, também é realizada uma avaliação do desempenho e breve descrição da implementação de paralelismo de dados e GPGPU. Mais tarde, uma nova versão do Denoiser foi apresentada pelos mesmos autores em [Aldinucci et al. 2014]. Nesse novo trabalho, o paralelismo da aplicação Denoiser é estendido para o domínio de *stream* com o suporte da biblioteca FastFlow. Entretanto, essa implementação foca em arquiteturas heterogêneas (multi-núcleo e GPGPU) enquanto que esse artigo trabalha com arquiteturas homogêneas (multi-núcleo). Em comparação com esse trabalho, [Aldinucci et al. 2012] e [Aldinucci et al. 2014] não comparam sua implementação com outras interfaces de paralelismo e não avaliam a programabilidade.

3. Denoiser

Denoiser é uma filtro capaz de restaurar imagens poluídas por esparsos píxeis pretos ou brancos (*salt and pepper*). O algoritmo é proposto em duas etapas: 1) identificação dos píxeis corrompidos e 2) restauração da imagem. Desta forma, Denoiser garante alta qualidade de restauração de imagem/vídeo com até 90% de ruído. Este estudo aborda paralelismo apenas em uma *stream* de vídeo. Na Figura 1, pode-se visualizar os resultados da execução e o fluxo de operação da aplicação. Inicialmente, a operação $Capture$ obtêm uma imagem de um vídeo. Se não possuir ruído, a imagem é então fornecida para o estágio $Noise$, que aleatoriamente aplicará o ruído *Salt and Pepper* nos píxeis da imagem. Na sequência, a imagem é enviada para o algoritmo $Detect$, onde os píxeis corrompidos são identificados e mapeados. Subsequentemente, os píxeis corrompidos da imagem são restaurados por $Denoise$ através de sucessivas iterações até a convergência. Finalmente, a imagem resultante é escrita em arquivo de saída na operação $Write$. Esse processo se repete para cada imagem do vídeo. O suporte para as operações de leitura e escrita em vídeo é fornecido pela biblioteca OpenCV.

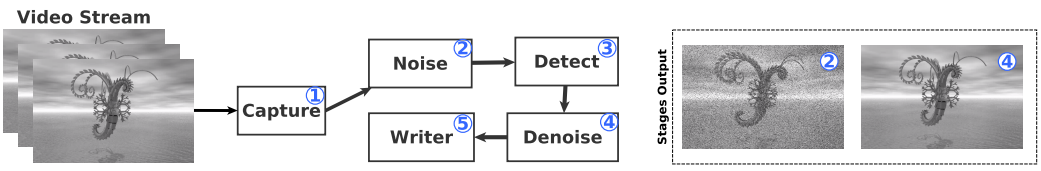


Figura 1. Fluxo de operação da aplicação Denoiser.

É possível adotar uma estratégia de paralelismo de dados, entretanto, desenvolvemos a versão paralela da aplicação Denoiser empregando elementos do paralelismo de *stream*. Dessa forma, apenas entradas de vídeo se beneficiam do paralelismo enquanto que imagens únicas são processadas sequencialmente. Essencialmente, a estrutura de paralelismo consiste em um *pipeline* de três estágios. Responsável por gerar os elementos da *stream*, o primeiro estágio é executado sequencialmente e encapsula as operações $Capture$, $Noise$ e $Detect$. Destas operações, apenas $Capture$ é estritamente sequencial. Entretanto, optamos por executar $Noise$ e $Detect$ no mesmo estágio. Isso porque, de acordo com nossos testes, essas duas operações representam uma parcela negligenciável da carga de trabalho total da aplicação. Mesmo sequencial, essas operações são capazes de gerar a vazão necessária para evitar um gargalo. Isso também foi constatado por [Aldinucci et al. 2014].

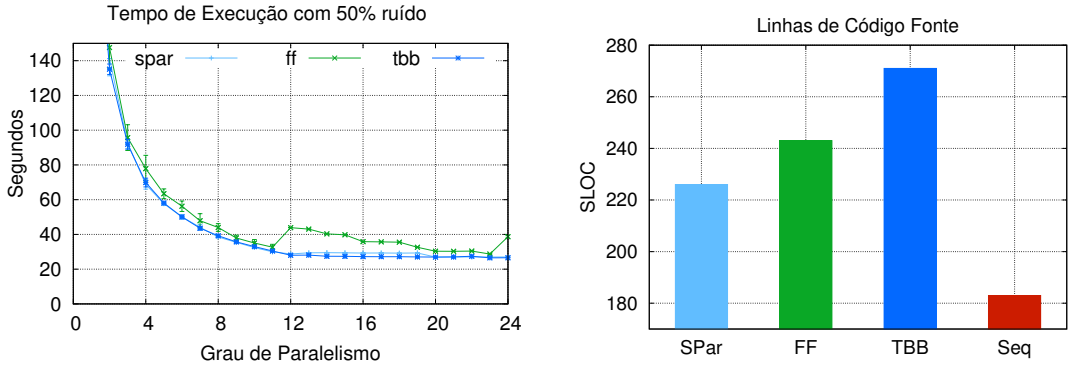
Envolto pelo segundo estágio do *pipeline*, o algoritmo $Denoise$ é a operação mais custosa da aplicação. Representando aproximadamente 95% do tempo de processamento de uma imagem, $Denoise$ é a operação onde o paralelismo é efetivado. Assim, diferentes imagens de um vídeo são processadas em paralelo de forma segura, visto que não possuem dependência de dados entre elas. Subsequentemente, o terceiro estágio do *pipeline* contém a operação $Write$, que é executada sequencialmente, uma vez que opera com apenas um canal de dados. Por padrão, em um programa paralelo, a ordem de processamento dos elementos da *stream* não é determinística. Consequentemente, foi necessário reordenar a *stream* no último estágio a fim de evitar a sobreposição no vídeo de saída.

4. Experimentos

Nesta seção, descrevemos os resultados obtidos e experimentos realizados. A máquina utilizada possui dois processadores *Intel(R) Xeon(R) CPU E5-2620 v3 2.40GHz* e 24GB de memória RAM. O sistema operacional foi Ubuntu Server 64 bits (*kernel 4.4.0-59-generic*). As ferramentas utilizadas foram: GCC (5.4.0), bibliotecas TBB (4.4 20151115), FastFlow (revisão 13) e OpenCV (2.4.9.1). Além disso, a compilação foi feita com a diretiva -O3. O vídeo de entrada é de formato AVI com tamanho de 7,4 MB (768x512 píxeis), que é carga padrão da aplicação base. Os valores foram gerados a partir da média aritmética de 10 execuções. O desvio padrão também é apresentado no gráfico. Para a avaliação de programabilidade, foi levado em consideração a métrica SLOC (número de linhas de código fonte). Todos os testes foram realizados para 3 versões: SPar (*spar*), FastFlow (*ff*) e TBB (*tbb*).

Apresentamos o Tempo de Execução de Denoiser (50% de ruído) na Figura 2(a). O tempo de execução sequencial é de 251 segundos. Aqui, observamos que a escalabilidade máxima do programa é atingida no grau de paralelismo 12 (número de *threads* físicas). Contudo, *ff* atinge esse ponto com apenas 11. Isso porque o FastFlow dedica uma *thread* exclusivamente para manejar o pipeline. Ainda, há um desbalanceamento de carga em *ff*, acentuado ao atingir o uso do recurso de *hyper-threading*. Esse desbalanceamento é solucionado em *spar* através da utilização de uma política de escalonamento sob-demanda em conjunto com um reordenamento livre de travas de sincronização. Já em *tbb*, o desbalanceamento de carga é lidado pelo *greedy-scheduling*, que impede que *threads* assumam um estado ocioso enquanto houver trabalho pendente.

O gráfico da Figura 2(b) apresenta o SLOC (Linhas de Código Fonte) das versões



(a) Tempo de Execução.

(b) SLOC.

Figura 2. Experimentos Denoiser.

avaliadas. Dessa forma, esperamos obter um indicativo de programabilidade avaliando a intrusão de código necessária para expressar o paralelismo da aplicação Denoiser. *Seq* representa a versão sequencial do código. Como esperado, a SPar obteve o menor percentual de aumento. Isso porque seu sistema de anotações permite manter a estrutura do código fonte razoavelmente inalterada. Por outro lado, as bibliotecas FF e TBB possuem um aumento de SLOC maior visto que necessitam refatorar o código sequencial e adicionar novas estruturas de dados para comunicação entre os estágios.

5. Conclusões

Este artigo apresentou uma análise de desempenho, programabilidade e paralelização da aplicação de restauração de imagem Denoiser. Os resultados indicam que, das interfaces avaliadas, TBB apresenta o menor tempo de execução enquanto que a SPar fica 6% atrás no pior dos casos. Já na avaliação programabilidade, a SPar possui aproximadamente 24% menos intrusão de código que o TBB e 9% para o FastFlow. Futuramente, pretendemos melhorar a geração de código da SPar com os resultados observados, além de estender os estudos da aplicação Denoiser através de novos testes com diferentes cargas de trabalho.

Referências

[Aldinucci et al. 2014] Aldinucci, M., Peretti Pezzi, G., Drocco, M., Tordini, F., Kilpatrick, P., and Torquati, M. (2014). Parallel video denoising on heterogeneous platforms. In *2014 HLPGPU*, pages 1–8, Vienna, Austria. HiPEAC.

[Aldinucci et al. 2012] Aldinucci, M., Spampinato, C., Drocco, M., Torquati, M., and Palazzo, S. (2012). A parallel edge preserving algorithm for salt and pepper image denoising. In *2012 IPTA*, pages 97–102, Istambul, Turkey. IEEE.

[Griebler et al. 2017] Griebler, D., Danelutto, M., Torquati, M., and Fernandes, L. G. (2017). SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):20.

[Reinders 2007] Reinders, J. (2007). *Intel Threading Building Blocks*. O'Reilly, Sebastopol, CA, USA.

[Wan et al. 2010] Wan, J., Ye, L., and Zhang, Q. (2010). Parallel multi-regions image restoration system and its implementation. In *2010 ICCSIT*, pages 216–220. IEEE.