Projeto Final MC970 - Paralelização da Técnicas de Meios-Tons

Vinicius P. M. Miguel - RA260731
Henrique Pareda de Souza - RA260497
Raphael Salles Vitor de Souza - RA223641

29 de junho de 2025

Sumário

1 Introdução

A técnica de meios-tons (halftone) consiste na criação de padrões formados por pontos pretos e brancos para reduzir a quantidade de níveis de cinza de uma imagem monocromática. Este método é amplamente empregado por veículos de comunicação impressos, como o jornal (Figura 1), onde podemos representar diversas tonalidades de cinza utilizando apenas tinta preta disposta em forma de círculos de raio variado.

Em meios digitais, o halftone pode ser adaptado para transformar uma imagem monocromática de 256 níveis de cinza para uma com apenas pixels pretos e brancos, sendo muito utilizado para visualização de imagens e impressões.

Além dessas aplicações clássicas, esta técnica pode ser empregada, por exemplo, na criação de mensagens criptografadas [2] e de marcas d'água ocultas em conteúdos autorais [3].

Neste contexto, este exercício visa aplicar algumas técnicas de meios-tons por difusão de erro muito difundidas na literatura, explorando e discutindo as principais nuances de cada abordagem.

2 Formulação

2.1 Formulação

Dada uma imagem de entrada A com $[a_{min}, ..., a_{max}]$ níveis de cinza, desejamos aplicar um algoritmo de halftone que construa uma nova imagem B preta e branca, de forma que a imagem transformada fique visualmente parecida com a entrada. Este processo pode ser aplicado tanto para imagens em preto e branco quanto para imagens coloridas, sendo que, no caso das coloridas, o algoritmo é executado separadamente para cada canal de cor.

Este conteúdo está em desenvolvimento. Planejamos alterar a forma de percorrer a matriz para implementar um pipeline que permita que este código seja mais eficiente em GPUs.

Figura 1: Aplicação de Halftone em materiais jornalísticos [1]

A matriz de difusão armazena o peso que um determinado pixel possui na distribuição do erro. Para cada valor na matriz de difusão, atualizamos A de acordo com a Equação 1, onde x' e y' representam cada coordenada da matriz de difusão. Os métodos que serão apresentados diferem apenas quanto à escolha da matriz de difusão.

$$A(x',\ y') \ = \ A(x',\ y') + (x',\ y') \ \cdot \ erro \tag{1}$$

No escopo deste projeto, serão abordados 6 métodos de difusão de erros distintos: Floyd e Steinberg; Stevenson e Arce; Burkes; Sierra; Stucki; e Jarvis, Judice e Ninke. As suas respectivas matrizes de difusão de erro estão apresentadas nas Tabelas 1 a 6, onde $f(x, y)$ corresponde a $A(x, y, canal)$.

	$f(x, y)$	7 / 16
3 / 16	5 / 16	1 / 16

Tabela 1: Floyd e Steinberg

			$f(x, y)$		32 / 200	
12 / 200		26 / 200		30 / 200		16 / 200
	12 / 200		26 / 200		12 / 200	
5 / 200		12 / 200		12 / 200		5 / 200

Tabela 2: Stevenson e Arce

		$f(x, y)$	8 / 32	4 / 32
2 / 32	4 / 32	8 / 32	4 / 32	2 / 32

Tabela 3: Burkes

		$f(x, y)$	5 / 32	3 / 32
2 / 32	4 / 32	5 / 32	4 / 32	2 / 32
	2 / 32	3 / 32	2 / 32	

Tabela 4: Sierra

		$f(x, y)$	8 / 42	4 / 42
2 / 42	4 / 42	8 / 42	4 / 42	2 / 42
1 / 42	2 / 42	4 / 42	2 / 42	1 / 42

Tabela 5: Stucki

		$f(x, y)$	7 / 48	5 / 48
3 / 48	5 / 48	7 / 48	5 / 48	3 / 48
1 / 48	3 / 48	5 / 48	3 / 48	1 / 48

Tabela 6: Jarvis, Judice e Ninke

2.2 Implementação

2.2.1 Execução

O código referente à implementação do halftone pode ser encontrado em *1_ halftoning.py*, recebendo como parâmetro no terminal o caminho até a entrada. O programa aceita mais de uma imagem de entrada, basta passar os respectivos caminhos em sequência.

3 Implementação

Neste capítulo, apresentamos as três versões do código desenvolvidas para fins comparativos: uma versão serial, uma versão paralelizada utilizando OpenMP e uma versão implementada em CUDA. Cada uma dessas versões será explicada em detalhes nas subseções a seguir.

3.1 Versão Serial

A versão serial do código, localizada no arquivo `serial.cpp`, foi implementada como ponto de partida para o desenvolvimento das versões paralelas. Nesta implementação, o algoritmo é executado de forma sequencial, processando os dados de entrada um elemento por vez. Essa abordagem é simples e direta, mas não aproveita os recursos de paralelismo disponíveis em arquiteturas modernas.

O código foi estruturado para ser claro e modular, facilitando a compreensão e a posterior paralelização. Ele realiza as seguintes etapas principais:

- Leitura dos dados de entrada.
- Processamento sequencial dos dados, aplicando a técnica de meios-tons utilizando difusão de erro.
- Escrita dos resultados no arquivo de saída.

O algoritmo de difusão de erro utilizado na versão serial é configurado para suportar diferentes métodos de dithering, como Floyd-Steinberg, Burkes, Sierra, entre outros. A escolha do método é feita com base em um parâmetro de entrada, permitindo flexibilidade na execução.

Essa versão serve como base para medir o desempenho inicial e comparar os ganhos obtidos com as versões paralelas. O código completo da implementação pode ser encontrado abaixo:

3.2 Versão Paralela com OpenMP

A versão paralela utilizando OpenMP foi desenvolvida a partir da versão serial, com o objetivo de explorar o paralelismo em CPUs multicore. O código-fonte desta versão é essencialmente o mesmo da versão serial, diferindo apenas pela inclusão de diretivas de paralelização do OpenMP, que são ativadas em tempo de compilação. Os resultados e o impacto dessas alterações serão discutidos na Seção 4.

3.3 Versão em CUDA

A versão em CUDA, que será implementada no futuro, tem como objetivo explorar o paralelismo massivo oferecido por GPUs. Esta subseção será preenchida com os detalhes da implementação assim que o código estiver concluído.

3.4 Implementação de Métodos Estocásticos

Além das versões mencionadas anteriormente, também foram implementados métodos estocásticos para avaliar como o uso de aleatoriedade pode impactar o resultado visual da técnica de meios-tons. Esses métodos introduzem variações aleatórias no processo de difusão de erro, com o objetivo de verificar se a qualidade visual das imagens processadas é significativamente alterada.

A implementação dos métodos estocásticos foi realizada adicionando um fator de ruído controlado ao cálculo da difusão de erro. Esse fator é gerado utilizando uma distribuição uniforme, garantindo que o comportamento do algoritmo permaneça previsível e reprodutível. As etapas principais dessa abordagem são as seguintes:

- Geração de um valor aleatório para cada pixel, dentro de um intervalo predefinido.
- Ajuste do erro difundido com base no valor aleatório gerado.
- Continuação do processamento com o erro ajustado.

Para as versões paralelas, foram adicionadas flags específicas para habilitar o suporte aos métodos estocásticos:

- Na versão OpenMP, a flag `-stochastic` foi utilizada durante a compilação para ativar o código relacionado aos métodos estocásticos. O comportamento estocástico pode ser controlado por um parâmetro de entrada, permitindo escolher entre dithering estocástico (1) ou padrão (0), sendo o padrão o valor 1.
- Na versão CUDA, a flag `-stochastic` também foi empregada, garantindo que o comportamento estocástico fosse incorporado ao kernel CUDA. Assim como na versão OpenMP, o parâmetro de entrada permite alternar entre dithering estocástico (1) ou padrão (0), com o valor padrão definido como 1.

Apesar de introduzir aleatoriedade no processo, os resultados preliminares indicam que o impacto visual é mínimo, especialmente em imagens com alta resolução. Isso sugere que os métodos estocásticos podem ser uma alternativa viável para cenários onde a uniformidade do padrão de dithering não é uma prioridade.

A análise detalhada dos resultados obtidos com os métodos estocásticos será apresentada na Seção 4, onde compararemos a qualidade visual e o desempenho em relação às outras versões do código.

3.5 Análise de WSNR

Para avaliar a qualidade das imagens processadas pelas diferentes versões do código, utilizamos o script `wsnr.py`, que calcula a métrica WSNR (Weighted Signal-to-Noise Ratio). Essa métrica é amplamente utilizada para medir a qualidade de imagens processadas, levando em consideração o ruído introduzido durante o processamento.

A seguir, apresentamos os resultados obtidos para cada método (serial, OpenMP e CUDA), organizados por tamanho de imagem:

Tabela de Resultados WSNR por Método e Tamanho de Imagem

Tamanho	Imagem	CUDA (dB)	OpenMP (dB)	Serial (dB)
Small	small_1	0.6595	-0.1043	-0.1279
	small_2	1.1523	0.6237	0.5631
Medium	medium_1	1.2968	0.8956	0.9051
	medium_2	1.2322	0.9434	0.8777
	medium_3	0.6089	0.2502	0.2566
	medium_4	0.7146	0.3194	0.2924
	medium_5	1.7778	0.7717	0.7879
	medium_6	1.3159	0.8904	0.8921
	medium_7	1.2963	0.7031	0.6980
	medium_8	-0.2986	-0.8768	-0.8693
Large	large_1	0.0847	-0.0862	-0.0905
	large_2	1.6083	0.8550	0.8642
	large_3	1.6087	0.8773	0.8754
	large_4	0.7675	0.7062	0.6588
	large_5	-0.3377	-0.8666	-0.8857
Xlarge	xlarge_1	1.6322	0.8531	0.8586
	xlarge_2	1.6904	0.9132	0.9014
	xlarge_3	1.6817	0.8648	0.8632
	xlarge_4	1.6876	0.9090	0.9010
	xlarge_5	0.7856	0.4889	0.4997
	xlarge_6	2.3256	1.3715	1.3500
	xlarge_7	0.8412	0.1943	0.2099
Média WSNR		**1.0969**	**0.5226**	**0.5128**

Tabela 7: Resultados WSNR por Método e Tamanho de Imagem

Interpretação dos Resultados

A métrica WSNR indica a relação entre o sinal e o ruído introduzido no processamento das imagens. Valores mais altos de WSNR indicam menor degradação da qualidade da imagem. Observa-se que a versão CUDA apresenta a maior média de WSNR (1.0969 dB), seguida pela versão OpenMP (0.5226 dB) e pela versão serial (0.5128 dB). Isso sugere que a versão CUDA não apenas é mais eficiente em termos de desempenho, mas também preserva melhor a qualidade das imagens processadas.

4 Speedup

Nesta seção, apresentamos as métricas de Speedup obtidas durante os experimentos. O Speedup é calculado como a razão entre o tempo de execução da versão serial e o tempo de execução das versões paralelas (OpenMP e CUDA). Foram consideradas duas abordagens principais:

- **Speedup por Método:** Para cada método de dithering (*FloydSteinberg*, *StevensonArce*, *Burkes*, *Sierra*, *Stucki*, *JarvisJudiceNinke*), calculamos o Speedup médio considerando todas as imagens processadas.
- **Speedup por Tamanho de Imagem:** Para cada tamanho de imagem (*e.g.*, 1920x1080, 1280x720), calculamos o Speedup médio considerando todos os métodos de dithering aplicados. Os tamanhos de imagem foram classificados em quatro categorias:
  - **Pequeno (small):** Imagens com largura e altura menores ou iguais a 256 pixels.
  - **Médio (medium):** Imagens com largura e altura menores ou iguais a 512 pixels.
  - **Grande (large):** Imagens com largura e altura menores ou iguais a 1024 pixels.
  - **Muito Grande (xlarge):** Imagens com largura ou altura maiores que 1024 pixels.

Os resultados obtidos estão destacados nas Tabelas 8 e 9. Os campos em vermelho indicam os valores que ainda precisam ser preenchidos com os dados experimentais.

Tabela 8: Resultados de Speedup por Método

Método	Serial (s)	OpenMP (s)	CUDA (s)	Speedup OpenMP	Speedup CUDA
FloydSteinberg	x.xx	x.xx	x.xx	x.xx	x.xx
StevensonArce	x.xx	x.xx	x.xx	x.xx	x.xx
Burkes	x.xx	x.xx	x.xx	x.xx	x.xx
Sierra	x.xx	x.xx	x.xx	x.xx	x.xx
Stucki	x.xx	x.xx	x.xx	x.xx	x.xx
JarvisJudiceNinke	x.xx	x.xx	x.xx	x.xx	x.xx

Tabela 9: Resultados de Speedup por Tamanho de Imagem

Tamanho da Imagem	Serial (s)	OpenMP (s)	CUDA (s)	Speedup OpenMP	Speedup CUDA
800x600	x.xx	x.xx	x.xx	x.xx	x.xx
1280x720	x.xx	x.xx	x.xx	x.xx	x.xx
1920x1080	x.xx	x.xx	x.xx	x.xx	x.xx

Tabela 10: Resultados de Eficiência por Método

Método	Eficiência OpenMP	Eficiência CUDA
FloydSteinberg	x.xx	x.xx
StevensonArce	x.xx	x.xx
Burkes	x.xx	x.xx
Sierra	x.xx	x.xx
Stucki	x.xx	x.xx
JarvisJudiceNinke	x.xx	x.xx

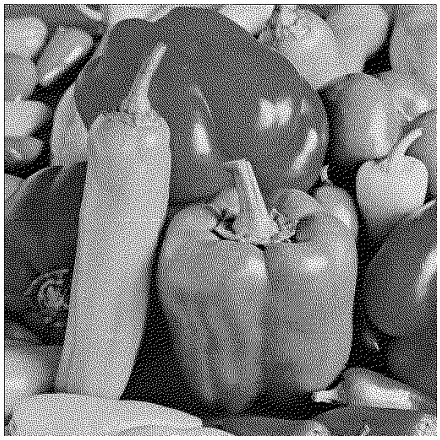
Tabela 11: Resultados de Eficiência por Tamanho de Imagem

Tamanho da Imagem	Eficiência OpenMP	Eficiência CUDA
800x600	x.xx	x.xx
1280x720	x.xx	x.xx
1920x1080	x.xx	x.xx

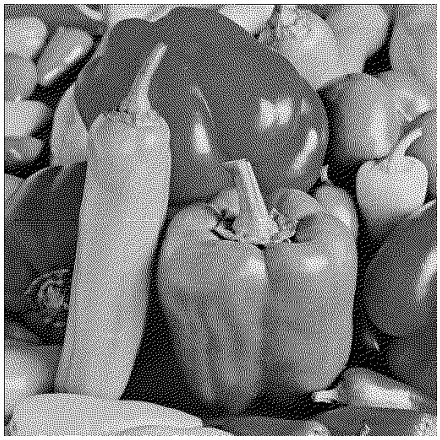
4.1 Método estocástico

Os códigos de OpenMP e CUDA possuíam uma flag que permitia a utilização de um método estocástico. Esse método introduz variações aleatórias no processo de dithering, o que pode impactar os resultados de Speedup e Eficiência. Durante os experimentos, observamos que o uso do método estocástico gerou diferenças nos tempos de execução e nos resultados finais. No entanto, como o objetivo do processo de dithering é criar um efeito visual na imagem, essas variações não comprometem a qualidade visual do resultado, tornando válida a sua utilização.

(a) Imagem Original



(b) OpenMP



(c) OpenMP Estocástico

Figura 2: Comparação visual entre as imagens: original, OpenMP e OpenMP com método estocástico.

Conferir se a implementacao do estocástico está correta

Os resultados obtidos com o método estocástico foram considerados satisfatórios. Apesar das variações introduzidas no processo de dithering, a qualidade visual das imagens geradas permaneceu consistente com os objetivos do experimento. A Figura 2 ilustra que as diferenças visuais entre as imagens processadas com e sem o método estocástico são mínimas, reforçando a validade do uso dessa abordagem nos experimentos realizados.

Tabela 12: Resultados de Speedup para o Método Estocástico

Método	**Serial (s)**	**OpenMP (s)**	**CUDA (s)**	**Speedup OpenMP**	**Speedup CUDA**
FloydSteinberg	x.xx	x.xx	x.xx	x.xx	x.xx
StevensonArce	x.xx	x.xx	x.xx	x.xx	x.xx
Burkes	x.xx	x.xx	x.xx	x.xx	x.xx
Sierra	x.xx	x.xx	x.xx	x.xx	x.xx
Stucki	x.xx	x.xx	x.xx	x.xx	x.xx
JarvisJudiceNinke	x.xx	x.xx	x.xx	x.xx	x.xx

Tabela 13: Resultados de Eficiência para o Método Estocástico

Método	**Eficiência OpenMP**	**Eficiência CUDA**
FloydSteinberg	x.xx	x.xx
StevensonArce	x.xx	x.xx
Burkes	x.xx	x.xx
Sierra	x.xx	x.xx
Stucki	x.xx	x.xx
JarvisJudiceNinke	x.xx	x.xx

5 Conclusão

Referências

[1] C. Sperandio, "Setting the right tone." `https://www.retrosupply.co/blogs/tutorials/setting-the-right-tone`.

[2] Z. Wang, G. R. Arce, and G. Di Crescenzo, "Halftone visual cryptography via error diffusion," *IEEE Transactions on Information Forensics and Security*, vol. 4, no. 3, pp. 383–396, 2009.

[3] M. S. Fu and O. Au, "Data hiding watermarking for halftone images," *IEEE Transactions on Image Processing*, vol. 11, no. 4, pp. 477–484, 2002.