

# 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

30 de junho de 2025

## Sumário

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Formulação</b>	<b>2</b>
2.1	Difusão de Erro . . . . .	2
2.2	Difusão de Erro Estocástica . . . . .	3
<b>3</b>	<b>Implementação</b>	<b>4</b>
3.1	Versão Serial . . . . .	4
3.2	Versão Paralela com OpenMP . . . . .	5
3.3	Versão em CUDA . . . . .	5
3.4	Implementação de Métodos Estocásticos . . . . .	6
<b>4</b>	<b>Benchmarking</b>	<b>7</b>
4.1	Ambiente de Teste . . . . .	7
4.2	Resultados . . . . .	8
4.3	Método estocástico . . . . .	8
4.4	Análise de WSNR . . . . .	10
<b>5</b>	<b>Conclusão</b>	<b>10</b>
	<b>Referências</b>	<b>12</b>

## 1 Introdução

A técnica de meios tons (*halftone*) consiste na redução de uma imagem monocromática de 256 níveis de cinza para pixels brancos e pretos, de forma que a imagem resultante fique visualmente parecida com a entrada. Um exemplo de aplicação dessa técnica pode ser visto na Figura 1, em que a imagem matém a estrutura visual, porém com menos níveis de informação. Isto é útil, por exemplo, em cenários em que deseja-se economizar armazenamento, com pouco prejuízo de perda de informação visual. Este método também pode ser utilizado em imagens coloridas, onde a técnica é aplicada separadamente em cada canal de cor.



Imagem original



Imagem com meios tons

Figura 1: Comparação entre a imagem original e a imagem após aplicação da técnica de meios tons.

O presente trabalho tem como objetivo a paralelização de algoritmos de meios tons baseados em difusão de erro. A difusão de erro é uma técnica que distribui o erro de quantização dos pixels vizinhos, permitindo uma melhor representação visual da imagem original. A paralelização desses algoritmos pode melhorar significativamente o desempenho, especialmente em imagens de alta resolução.

Estabelece-se, portanto, como objetivo específico deste trabalho a comparação do desempenho entre as versões serial, OpenMP e CUDA de seis algoritmos consagrados na literatura: Floyd & Steinberg; Steverson & Arce; Burkes; Sierra; Stucki; e Jarvis Judice & Ninke.

## 2 Formulação

### 2.1 Difusão de Erro

Dada uma imagem com 255 níveis de cinza, nosso objetivo é aplicar um algoritmo que gere uma nova imagem contendo apenas pixels pretos e brancos, de modo que o resultado final seja visualmente semelhante à imagem original. Esse procedimento pode ser realizado tanto em imagens monocromáticas quanto coloridas. No caso de imagens coloridas, o algoritmo é aplicado individualmente a cada canal de cor, convertendo o valor de cada pixel em cada canal para preto (0) ou para o valor máximo (255). O procedimento geral pode ser encontrado no Algoritmo 1.

---

**Algorithm 1** Algoritmo Geral de Halftone por Difusão de Erro

---

```

1: for all canal  $\in$  Imagem A do
2:   for  $x = 0$  até  $m - 1$  do
3:     for  $y = 0$  até  $n - 1$  do
4:       if  $A(x, y, \text{canal}) < 128$  then
5:          $B(x, y, \text{canal}) = 0$ 
6:       else
7:          $B(x, y, \text{canal}) = 255$ 
8:       end if
9:
10:      erro =  $A(x, y, \text{canal}) - B(x, y, \text{canal})$ 
11:      DistribuirErro( $A$ , matrizDifusao, erro)
12:    end for
13:  end for
14: end for

```

---

As matrizes de difusão de erro são estruturas que determinam como o erro de quantização de um pixel é distribuído para seus vizinhos ainda não processados. Cada método de difusão utiliza uma matriz diferente, cujos valores representam a fração do erro a ser repassada para cada pixel vizinho. O objetivo é espalhar o erro de forma controlada, minimizando artefatos visuais e mantendo a aparência da imagem original.

Por exemplo, na matriz de Floyd e Steinberg, o erro é distribuído para quatro pixels vizinhos, com diferentes pesos, enquanto métodos como Jarvis, Judice e Ninke distribuem o erro para um número maior de vizinhos, suavizando ainda mais a transição dos tons. A escolha da matriz influencia diretamente o resultado visual do halftone e o desempenho do algoritmo.

$$A(x, y) = \text{matrizDifusao}(x, y) + (x, y) \cdot \text{erro} \quad (1)$$

Neste projeto, analisamos seis métodos clássicos de difusão de erro: Floyd e Steinberg, Stevenson e Arce, Burkes, Sierra, Stucki e Jarvis, Judice e Ninke. As matrizes de difusão associadas a cada método estão apresentadas nas Tabelas 1 a 6, onde  $f(x, y)$  representa o pixel corrente  $A(x, y, \text{canal})$ . Cada matriz define como o erro de quantização é distribuído entre os vizinhos, influenciando diretamente o resultado visual do halftone.

	$f(x, y)$	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 Difusão de Erro Estocástica

O método estocástico de difusão de erro [1] introduz um elemento de aleatoriedade no processo tradicional de dithering. Diferente do método determinístico, que distribui o erro de quantização com base em uma matriz fixa de pesos, o método

estocástico modifica a quantidade de erro difundido com base em um fator aleatório. Essa abordagem visa reduzir padrões visuais repetitivos e suavizar artefatos que podem surgir em regiões uniformes da imagem.

A implementação consiste em adicionar um ruído aleatório controlado ao valor do erro antes de sua distribuição. Formalmente, seja  $e$  o erro de quantização de um pixel, e  $r \sim \mathcal{U}(-p, p)$  um valor extraído de uma distribuição uniforme, com  $p \in [0, 1]$  sendo um parâmetro de controle da aleatoriedade. O erro ajustado  $e'$  é então dado por:

$$e' = e \cdot (1 + r)$$

Esse erro ajustado  $e'$  é então distribuído normalmente conforme a matriz de difusão escolhida (por exemplo, Floyd-Steinberg), mantendo a estrutura do algoritmo original. O valor de  $p$  pode ser ajustado para controlar a intensidade da estocasticidade. Valores menores de  $p$  resultam em comportamento mais próximo ao determinístico, enquanto valores maiores aumentam a variação aleatória.

## 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. A figura 2 ilustra a função principal de difusão de erro na versão serial, onde cada pixel é processado individualmente, aplicando a técnica de meios-tons com base no erro de quantização.

```
void apply_error_diffusion_serial(float* image, int width, int height, const std::map<std::pair<int, int>, float>& weights) {
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            int idx = y * width + x;
            float old_pixel = image[idx];
            float new_pixel = old_pixel < 0.5f ? 0.0f : 1.0f;
            float quant_error = old_pixel - new_pixel;
            image[idx] = new_pixel;

            for (const auto& [offset, weight] : weights) {
                int dx = offset.first;
                int dy = offset.second;
                int nx = x + dx;
                int ny = y + dy;
                if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
                    image[ny * width + nx] += quant_error * weight;
                }
            }
        }
    }
}
```

Figura 2: Função principal de difusão de erro na versão serial.

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 todos os métodos descritos na Subseção 2.1. 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.

## 3.2 Versão Paralela com OpenMP

A versão paralela utilizando OpenMP, implementada no arquivo `omp.cpp`, foi desenvolvida a partir da versão serial com o objetivo de explorar o paralelismo disponível em CPUs multicore. Nesta implementação, o processamento dos pixels é realizado de forma concorrente, utilizando diretivas OpenMP para paralelizar os principais laços do algoritmo. Isso permite uma redução significativa no tempo de execução, especialmente para imagens de maior resolução.

O código mantém a estrutura modular da versão serial, facilitando a manutenção e a comparação entre as abordagens. Além disso, a versão OpenMP suporta tanto o processamento de imagens em escala de cinza quanto coloridas, aplicando a difusão de erro separadamente em cada canal de cor. A figura 3 ilustra a função principal de difusão de erro na versão OpenMP.

```
void apply_error_diffusion_omp(float* image, int width, int height, const std::map<std::pair<int, int>, float>& weights) {
    #pragma omp parallel for
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            int idx = y * width + x;
            float old_pixel = image[idx];
            float new_pixel = old_pixel < 0.5f ? 0.0f : 1.0f;
            float quant_error = old_pixel - new_pixel;
            image[idx] = new_pixel;

            for (const auto& [offset, weight] : weights) {
                int dx = offset.first;
                int dy = offset.second;
                int nx = x + dx;
                int ny = y + dy;
                if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
                    image[ny * width + nx] += quant_error * weight;
                }
            }
        }
    }
}
```

Figura 3: Função principal de difusão de erro na versão OpenMP.

Novamente, o modo de difusão pode ser selecionado por meio dos parâmetros de entrada. O desempenho e o impacto visual da versão OpenMP são discutidos em detalhes na Seção 4.

## 3.3 Versão em CUDA

A versão CUDA, implementada no arquivo `cuda.cu`, foi desenvolvida para explorar o paralelismo massivo oferecido por GPUs. Assim como nas versões anteriores, o código é capaz de processar imagens em escala de cinza ou coloridas, aplicando a difusão de erro separadamente em cada canal de cor.

Nesta implementação, o processamento dos pixels é realizado de forma paralela ao longo das diagonais da imagem, garantindo que a propagação do erro respeite as dependências espaciais do algoritmo. Cada thread da GPU é responsável por processar um pixel em uma determinada diagonal, e a difusão do erro para os vizinhos é feita utilizando operações atômicas para evitar condições de corrida.

As principais etapas da versão CUDA são:

- Leitura dos dados de entrada e transferência da imagem para a memória da GPU.
- Processamento paralelo dos pixels ao longo das diagonais, aplicando a técnica de meios-tons com difusão de erro.
- Transferência dos resultados da GPU de volta para a memória do host.
- Escrita dos resultados no arquivo de saída.

As Figura 4 e 5 mostram, respectivamente, o *kernel* de processamento de diagonais e a função principal de difusão de erro na versão CUDA.

```

__global__ void process_diagonal(float* image, int diag, int width, int height,
                                const int* dx, const int* dy, const float* coef, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int x, y;
    get_coordinates_from_diagonal(idx, diag, width, height, &x, &y);
    if (x >= 0 && y >= 0 && x < width && y < height) {
        apply_method(image, x, y, width, height, dx, dy, coef, n);
    }
}

```

Figura 4: Função principal de difusão de erro na versão CUDA.

```

__device__ void apply_method(float* image, int x, int y, int width, int height,
                             const int* dx, const int* dy, const float* coef, int n) {
    int idx = y * width + x;
    float old_pixel = image[idx];
    float new_pixel = old_pixel < 0.5f ? 0.0f : 1.0f;
    float quant_error = old_pixel - new_pixel;
    image[idx] = new_pixel;

    for (int i = 0; i < n; ++i) {
        int nx = x + dx[i];
        int ny = y + dy[i];
        if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
            atomicAdd(&image[ny * width + nx], quant_error * coef[i]);
        }
    }
}

```

Figura 5: Função principal de difusão de erro na versão CUDA.

O método de difusão de erro utilizado pode ser selecionado por parâmetro de entrada, assim como nas demais versões. O desempenho e a qualidade visual dos resultados obtidos com a versão CUDA são discutidos na Seção 4.

### 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 reproduzível. 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.

## 4 Benchmarking

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 das subseções a seguir.

### 4.1 Ambiente de Teste

O Benchmarking foi realizado em um sistema com as seguintes especificações:

- **Processador:** Intel Core i7-11800H
- **Memória RAM:** 16 GB DDR4
- **Placa de Vídeo:** NVIDIA GeForce RTX 3060 80W
- **Versão do CUDA:** 12.9
- **Driver NVIDIA:** 575.64
- **Compilador C/C++:** GCC 15.1.1
- **Versão do OpenMP:** 4.5
- **Sistema Operacional:** Arch Linux (rolling release)
- **Versão do kernel:** 6.15.4-arch2-1

Especialmente na versão com OpenMP, como não foi especificado o número de threads, o código utilizou o número de núcleos lógicos disponíveis pelo processador, que nesse ambiente de testes, foram 16.

Para a coleta de dados, utilizamos o `omp_get_wtime()` para medir o tempo de execução dos códigos. O método de dithering aplicado é passado via linha de comando (como especificado no README do projeto). Todos os dados foram coletados utilizando *grayscale* ativado. O método de dithering base para medição de impacto de acordo com o tamanho da imagem é o *FloydSteinberg*. As imagens utilizadas foram obtidas do repositório oficial do projeto, conforme descrito no README.

Organizamos um script de benchmarking, que executa cada imagem 5 vezes, onde conseguimos alterar o método de dithering, flag de método estocástico e flag de escala de cinza. O script foi implementado em `colab-runner.ipynb` na raiz do repositório.

## 4.2 Resultados

Tabela 7: Resultados de Speedup por Método

Método	Serial (ms)	OpenMP (ms)	CUDA (ms)	Speedup OpenMP	Speedup CUDA
FloydSteinberg	126.52 $\pm$ 301.72	25.08 $\pm$ 47.89	146.27 $\pm$ 317.29	5.04 $\pm$ 15.41	0.86 $\pm$ 2.79
StevensonArce	311.94 $\pm$ 748.04	57.94 $\pm$ 122.09	146.80 $\pm$ 317.41	5.38 $\pm$ 12.91	2.12 $\pm$ 6.86
Burkes	191.00 $\pm$ 455.59	38.36 $\pm$ 82.35	146.64 $\pm$ 318.72	4.98 $\pm$ 15.98	1.30 $\pm$ 4.20
Sierra	262.05 $\pm$ 619.44	48.04 $\pm$ 102.86	145.37 $\pm$ 318.33	5.45 $\pm$ 17.40	1.80 $\pm$ 5.81
Stucki	309.84 $\pm$ 739.82	53.09 $\pm$ 112.06	145.15 $\pm$ 317.87	5.84 $\pm$ 21.18	2.13 $\pm$ 6.92
JarvisJudiceNinke	305.95 $\pm$ 720.93	56.04 $\pm$ 124.03	147.02 $\pm$ 317.46	5.46 $\pm$ 33.53	2.08 $\pm$ 6.65

A incerteza associada nos tempos de execução e no speedup foram bastante altas. Isso aconteceu pois na gama de testes realizados, as implementações receberam imagens de tamanhos variados (de small a xlarge). Por conta dessa imprecisão, a tabela 12 não inclui as incertezas, contentando apenas os valores médios de Speedup.

Tabela 8: Resultados de Speedup por Tamanho de Imagem

Tamanho da Imagem	Serial (ms)	OpenMP (ms)	CUDA (ms)	Speedup OpenMP	Speedup CUDA
Small	4.57 $\pm$ 0.78	3.81 $\pm$ 2.81	7.76 $\pm$ 0.95	1.20 $\pm$ 0.91	0.59 $\pm$ 0.12
Medium	21.20 $\pm$ 0.78	7.71 $\pm$ 3.25	30.65 $\pm$ 2.33	2.75 $\pm$ 1.16	0.69 $\pm$ 0.08
Large	37.90 $\pm$ 19.62	10.66 $\pm$ 4.04	58.64 $\pm$ 26.64	3.56 $\pm$ 2.06	0.65 $\pm$ 0.45
Extra Large	345.02 $\pm$ 486.16	61.30 $\pm$ 75.79	380.57 $\pm$ 507.17	5.63 $\pm$ 8.30	0.91 $\pm$ 1.76

Quando analisamos um em uma escala mais controlada de tamanho de imagem, vemos que as incertezas diminuem, ressaltando a análise sobre a tabela anterior. Vemos que o CUDA não apresenta um bom speedup e uma Occupancy extremamente baixa, o que pode ser explicado pelo fato de que nessa implementação, o dado possui dependência espacial, ou seja, apresenta baixo paralelismo.

Tabela 9: Resultados de Eficiência por Método

Método	Eficiência OpenMP	Occupancy CUDA (%)
FloydSteinberg	0.32	2.08
StevensonArce	0.34	2.08
Burkes	0.31	2.08
Sierra	0.34	2.08
Stucki	0.37	2.08
JarvisJudiceNinke	0.34	2.08

Utilizando as medições para o método *FloydSteinberg* como base, vamos analisar a eficiência do OpenMP em relação ao tamanho da imagem.

Tabela 10: Resultados de Eficiência do OpenMP por Tamanho de Imagem

Tamanho da Imagem	Eficiência OpenMP
Small	0.08
Medium	0.17
Large	0.22
Extra Large	0.35

## 4.3 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.

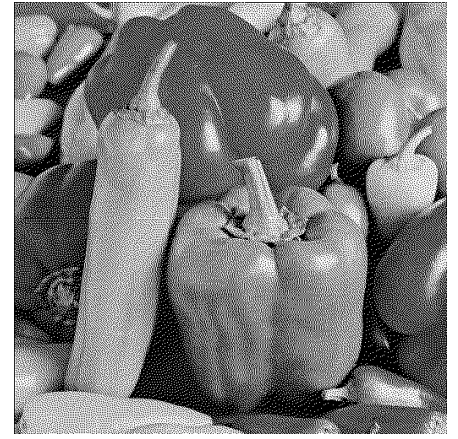




(a) Imagem Original



(b) OpenMP



(c) OpenMP Estocástico

Figura 6: Comparação visual entre as imagens: original, OpenMP e OpenMP com método estocástico.

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 6 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.

Diferente de antes, vamos apresentar os resultados sem levar em consideração à incerteza associada, a fim de evitar a redundância de informação sobre as diferenças dos tamanhos de imagens. É aceitável que os resultados sejam apresentados sem considerar a incerteza, uma vez que o foco é comparar os métodos de dithering e suas implementações paralelas de forma geral.

Tabela 11: Tempos Médios de Execução para o Método Estocástico

Método	Serial (ms)	OpenMP (ms)	CUDA (ms)
FloydSteinberg	795.46	110.73	0.42
StevensonArce	2220.53	298.43	0.54
Burkes	1310.33	125.78	0.45
Sierra	1852.19	246.34	0.49
Stucki	2204.74	284.48	0.52
JarvisJudiceNinke	2213.28	284.34	0.52

Como o tempo de execução do método estocástico do serial foi muito maior que a implementação comum, vamos utilizar a implementação não estocástica como base a tabela [7] para o cálculo do Speedup e Eficiência.

Tabela 12: Speedup para o Método Estocástico

Método	Speedup OpenMP	Speedup CUDA
FloydSteinberg	1.14	301.24
StevensonArce	1.04	577.67
Burkes	1.03	424.44
Sierra	1.06	534.80
Stucki	1.09	595.85
JarvisJudiceNinke	1.07	588.37

Tabela 13: Resultados de Eficiência para o Método Estocástico

Método	Eficiência OpenMP	Occupancy CUDA (%)
FloydSteinberg	0.07	61.65
StevensonArce	0.07	60.90
Burkes	0.06	59.68
Sierra	0.07	59.54
Stucki	0.07	59.90
JarvisJudiceNinke	0.07	61.01

Vemos que a Occupancy aumentou significativamente, e com isso, o speedup obtido com o CUDA foi muito maior que as outras implementações.

#### 4.4 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
<b>Média WSNR</b>		<b>1.0969</b>	<b>0.5226</b>	<b>0.5128</b>

Tabela 14: 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.

## 5 Conclusão

Este trabalho explorou diferentes estratégias para a paralelização de algoritmos de difusão de erro, com foco nos métodos clássicos da literatura. Foram desenvolvidas três versões: uma serial, uma paralela utilizando OpenMP e uma versão em CUDA, além da inclusão de uma abordagem estocástica que introduz aleatoriedade no processo de quantização.

A avaliação de desempenho dos métodos convencionais mostra que a paralelização com OpenMP é capaz de gerar speedups significativos em relação à versão serial, especialmente para imagens grandes. Em média, os speedups variaram de 4.9x a 5.8x, com destaque para o método Stucki (5.84x) e Sierra (5.45x). Já a versão CUDA, surpreendentemente, apresentou desempenho inferior ao esperado para GPUs. Isso se deve, principalmente, à baixa ocupação da GPU (2.08%) em todos os métodos, sugerindo subutilização dos recursos, sendo uma consequência do baixo paralelismo na implementação tradicional (forte dependência espacial entre os pixels).

Ao analisarmos por tamanho de imagem, o OpenMP teve eficiência crescente com a resolução, indo de 0.08 (imagens pequenas) até 0.35 (imagens extra grandes). Isso confirma que o OpenMP é mais eficaz em ambientes com grande volume de dados, mas seu ganho é limitado em imagens pequenas devido ao overhead de paralelização. Já o CUDA apresentou speedups abaixo de 1.0 em imagens pequenas e médias, atingindo no máximo 0.91x em imagens extra grandes, refletindo a ineficiência geral da sua implementação convencional.

O cenário muda completamente com os métodos estocásticos. Nessa abordagem, a implementação CUDA se destaca, apresentando speedups de mais de 500x, com ocupação da GPU superior a 59%. O método Stucki, por exemplo, obteve speedup de 595.85x, enquanto Jarvis Judice Ninke chegou a 588.37x, um salto em relação à versão determinística. Isso mostra que a estrutura de paralelismo da GPU foi muito melhor aproveitada na distribuição de erro estocástica, devido a redução de dependências e a melhor distribuição de carga.

Já o OpenMP estocástico teve desempenho significativamente inferior ao esperado, com speedups entre 1.03x e 1.14x, indicando que a inserção de aleatoriedade causou perda de paralelismo efetivo. Sua eficiência ficou em torno de 0.06 a 0.07, muito abaixo da eficiência obtida nos métodos convencionais.

No aspecto visual, a análise de WSNR mostrou que a versão CUDA apresentou a maior fidelidade média (1.10 dB), seguida por OpenMP (0.52 dB) e Serial (0.51 dB), sugerindo que o uso da GPU não só acelerou a execução, como preservou (ou até melhorou) a qualidade da imagem. A variação introduzida pelo método estocástico não comprometeu a aparência final, validando seu uso em aplicações práticas.

## Referências

- [1] G. Franchini, R. Cavicchioli, H. JC, *et al.*, “Automatic stochastic dithering techniques on gpu: Image quality and processing time improved,” *ADVANCES IN SCIENCE, TECHNOLOGY AND ENGINEERING SYSTEMS JOURNAL*, vol. 5, no. 6, pp. 653–663, 2020.