Relatório 4: Método de Convolução em Processamento de Imagem - Desfoque Gaussiano

Jonatan Felipe Hartmann
Universidade Federal de Santa Catarina
Ciências da Computação
Florianópolis, Brasil

Lucas de Souza Vieira
Universidade Federal de Santa Catarina
Ciências da Computação
Florianópolis, Brasil

Lucas Tomio Schwochow
Universidade Federal de Santa Catarina
Ciências da Computação
Florianópolis, Brasil

Pedro Henrique Gimenez
Universidade Federal de Santa Catarina
Ciências da Computação
Florianópolis, Brasil

Rodrigo Schwartz
Universidade Federal de Santa Catarina
Ciências da Computação
Florianópolis, Brasil

Victoria Rodrigues Veloso
Universidade Federal de Santa Catarina
Ciências da Computação
Florianópolis, Brasil

Resumo

Este relatório apresenta o desenvolvimento de um sistema baseado em VHDL utilizando Quartus II para processamento de imagens através da aplicação de um filtro de Desfoque Gaussiano. O projeto se concentra na implementação de uma operação de convolução entre a matriz do kernel do filtro e a matriz da imagem para obter o efeito de desfoque. A arquitetura VHDL executa a convolução em matrizes 3x3, gerando uma matriz modificada que representa a imagem borrada. O sistema desenvolvido visa aprimorar a compreensão e implementação de algoritmos de processamento de imagens dentro de uma estrutura de linguagem de descrição de hardware (VHDL).

O fluxo de trabalho começa com a importação de uma representação de imagem baseada em matriz, convertida anteriormente por um script Python, para o ambiente VHDL, seguida pela aplicação do algoritmo de Desfoque Gaussiano por meio de operações de convolução. O processo de convolução envolve multiplicação e soma de pixels, realizada iterativamente na matriz da imagem e na matriz do filtro. Após a conclusão da convolução, o sistema produz uma matriz modificada que captura o efeito desfocado.

Além disso, um script Python é utilizado para transformar a matriz resultante novamente em uma imagem JPG, permitindo a representação visual e a avaliação do desfoque aplicado.

***Palavras-chave*—VHDL, processamento de imagem, filtro de desfoque gaussiano, matriz de convolução, implementação de hardware, Quartus II, transformação de imagem**

I. Introdução

A convolução é uma importante operação matemática utilizada no processamento de sinais. Seu fundamento consiste em combinar duas funções, gerando uma terceira que representa a forma como uma modifica a outra. A figura 1 mostra a expressão geral da convolução aplicada no processamento de imagens.

$$g(x,y) = \omega * f(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} \omega(i,j) f(x-i, y-j),$$

Fig. 1. Expressão de convolução para aplicação de filtros em imagens
Fonte: wikipedia, 2023

O Desfoque Gaussiano é um filtro comumente utilizado para redução de irregularidades e suavização do valor do pixel de uma fotografia. Entre as principais vantagens de usar o Desfoque Gaussiano, destacam-se a facilidade de visualização e a eficiência em reduzir ruídos para suavizar uma imagem. Este filtro apresenta uma matriz, também conhecida como kernel, que está ilustrada na imagem 2.

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Fig. 2. Kernel para o filtro gaussianoFonte: wikipedia, 2023

Com isso, o presente trabalho tem como objetivo demonstrar a aplicação do filtro de desfoque Gaussiano em imagens, utilizando uma arquitetura que aplica o método de convolução, feita na linguagem de descrição de hardware VHDL. No documento, será apresentado como foi organizado o script Python para transformar uma imagem em uma matriz e como foram estruturados os componentes do sistema digital de aplicação de convolução entre kernel do filtro e imagem

II. Metodologia

A. Script Python

O script de conversão da imagem em uma matriz foi realizado na linguagem Python. Além da conversão mencionada, também foi utilizado para realizar a convolução, em que seu resultado será utilizado posteriormente no

testbench. Também foi codificada a função que transforma a matriz final em imagem de novo para a representação visual do resultado.

*1) **Conversor de Imagem para Matriz**:* O script utiliza a biblioteca *Pillow* que gerencia tratamentos de imagens. Inicia-se lendo a imagem recebida pela função. Após a leitura, a imagem é convertida para tons na escala cinza para facilitar o cálculo posterior, e é redimensionada para o tamanho desejado. O tamanho padrão da imagem final é 600x600 pois o sistema digital em VHDL será baseado neste tamanho. Após o redimensionamento, o valor de cada pixel em decimal é obtido. A faixa de valores dos pixels é de [0, 255], visto que a imagem possui apenas tons da escala de cinza, sendo 0 preto e 255 branco. Depois, é realizada a conversão dos valores de cada pixel, que estavam em decimais, para binários de 8 bits, e então, a matriz é criada. Com a matriz criada, é realizado um tratamento, onde os valores extremos da matriz (primeira linha, última linha, primeira coluna e última coluna) são copiados e inseridos ao redor da matriz, redimensionando-a para uma matriz 602x602. Esse tratamento visa evitar o acesso de posições não existentes no momento de aplicação do filtro e mantém um valor aproximado para as bordas da imagem. Por fim, é feita a leitura da matriz, percorrendo todas as colunas de cada linha, e salvando esses valores em um arquivo .txt, onde cada linha desse arquivo representa o valor em binário de 8 bits de um pixel da matriz da imagem. Assim, posteriormente, esse arquivo será utilizado como o arquivo de estímulos para o testbench do sistema de convolução descrito em VHDL.

*2) **Aplicação da Convolução**:* Para simular o que irá ser realizado em hardware, foi realizado um script em Python com uma estrutura próxima a de uma na linguagem VHDL, utilizando sinais e *while loops*. A função recebe como parâmetro o arquivo .txt da imagem convertida em matriz e o arquivo .txt do *kernel* em binário. O script começa salvando os dados recebidos em listas, uma para a imagem e outra para o *kernel*, em que seus valores são convertidos para decimal para o facilitamento do cálculo em Python. Em seguida, cria-se uma lista simulando a memória de saída em que todos os valores são "255" para serem posteriormente alterados durante o processamento. Começa-se um *while loop* que percorre as linhas da matriz e outro *while loop* dentro percorrendo cada coluna. Dentro de cada coluna, começa o *while loop* percorrendo a matriz 3x3 do *kernel* e do "corte" da imagem. Calcula-se a convolução de cada pixel somando os produtos da matriz com o kernel. Em seguida, o sistema armazena o valor resultante na posição desejada da matriz final. Após realizar as operações, a função salva a matriz em um arquivo "matrizfiltrada.txt" em que cada linha é um pixel. O arquivo final possui 600x600 linhas.

*3) **Conversor de Matriz para Imagem**:* Para visualizar o resultado final, foi criada uma função que recebe um arquivo .txt com a matriz e salva uma imagem .jpg do resultado. A função começa chamando uma outra função que realiza a leitura de um arquivo .txt e transforma seu conteúdo em uma matriz. Com a matriz extraída do arquivo, começa-se a percorrer cada elemento da lista e o transforma em um pixel por meio da biblioteca *Pillow*. Após cada pixel ser armazenado, a função salva a imagem final em um arquivo .jpg.

B. Projeto RTL

O desenvolvimento do sistema e dos componentes utilizados tiveram como base o pseudo algoritmo que está ilustrado em II-B, visando realizar as mesmas etapas e operações feitas nele para a aplicação da convolução, mas no contexto da linguagem VHDL. Conforme o circuito foi sendo modelado, algumas modificações foram realizadas, ou seja, o sistema digital final teve diversas mudanças em relação ao pseudocódigo base. O diagrama de blocos do projeto com as modificações pode ser visualizado nas referências [2]

Algoritmo para Processamento de Imagens
$linha \leftarrow 1; coluna \leftarrow 1; pronto \leftarrow 0; memsaida \leftarrow [vazio]; mem \leftarrow data; kernel \leftarrow data;$
{Carregar memória e kernel com o arquivo}
while $linha < 299$ **do**
  while $coluna < 299$ **do**
    $soma \leftarrow 0; i \leftarrow 0; deslocaLinha \leftarrow -1; deslocaColuna \leftarrow -1;$
    while $i < 9$ **do**
      $soma \leftarrow soma + (kernel[i] * mem[(linha + deslocaLinha) * 300 + (coluna + deslocaColuna)];$

      $soma \leftarrow soma << 4$ {DIVIDINDO POR 16 UNICAMENTE NO FILTRO GAUSSIANO}
      $i \leftarrow i + 1; deslocaColuna \leftarrow deslocaColuna + 1; memsaida[linha * 300 + coluna] \leftarrow soma;$
      if $deslocaColuna > 1$ **then**
        $deslocaColuna \leftarrow 0; deslocaLinha \leftarrow deslocaLinha + 1;$
      end if
    end while
    $pronto \leftarrow 1;$
  end while
end while
{Acessar memória: mem[linha*300 + coluna]}
{Memórias acessadas em cada iteração do kernel:}
$mem[(linha - 1) * 300 + (coluna - 1)]$
$mem[(linha - 1) * 300 + (coluna)]$
$mem[(linha - 1) * 300 + (coluna + 1)]$
$mem[(linha) * 300 + (coluna - 1)]$
$mem[(linha) * 300 + (coluna)]$
$mem[(linha) * 300 + (coluna + 1)]$
$mem[(linha + 1) * 300 + (coluna - 1)]$
$mem[(linha + 1) * 300 + (coluna)]$
$mem[(linha + 1) * 300 + (coluna + 1)]$
{Codificação de cada "variável" do algoritmo:}
$linha$: STD_LOGIC_VECTOR(8 downto 0)
$coluna$: STD_LOGIC_VECTOR(8 downto 0)
$pronto$: STD_LOGIC

$memsaida$: array (0 to 89999) of bit_vector(7 downto 0)
mem : array (0 to 89999) of bit_vector(7 downto 0)
$kernel$: array (0 to 8) of bit_vector(2 downto 0) {AQUI PRECISAMOS DEFINIR QUAL FILTRO USAREMOS, PENSAR TAMBÉM NA QUESTÃO DE QUE O FILTRO GAUSSIANO TEM QUE DIVIDIR TUDO POR 16 (ou usar filtros só com valores inteiros como o de detecção de borda).}
$soma$: STD_LOGIC_VECTOR(11 downto 0) {para o máximo que a soma precisa}
$deslocaLinha$: signed(2 downto 0)
$deslocaColuna$: signed(2 downto 0)
i : STD_LOGIC_VECTOR(3 downto 0)

1) ***Bloco operativo****:* Iniciou-se o projeto em nível RTL com a subdivisão entre o bloco operativo e o bloco de controle, com base na simulação do que deveria ser realizado. No bloco operativo (datapath), foi necessário desenvolver uma lógica para os contadores de linhas e colunas, uma vez que a imagem possui dimensões de 600x600 pixels. Contudo, uma borda adicional foi acrescentada, resultando em uma imagem final de 602x602 pixels (com um pixel extra tanto na borda esquerda quanto na borda direita). Na imagem X podemos observar um dos contadores criados. Todos eles seguiram a mesma lógica em nível rt, variando apenas o número de bits (10 bits para os contadores de linha e coluna) e o número que deve ser comparado (igual a 601 para linha e coluna e 9 para o kernel). Desconsiderando as bordas, o contador de colunas, por exemplo, verifica se chegamos à coluna 601.

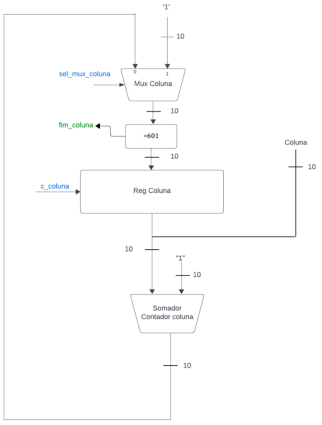

Fig. 3. Contador de colunas do bloco operativoFonte: autoria própria

Caso a afirmativa esteja correta, um sinal é enviado para que possamos pular para a próxima coluna.

Com o sinal dos contadores enviado, o endereço de memória foi calculado para que o dado seja acessado corretamente. Na imagem K observamos o esquema para calcular o acesso a memória mem, que é responsável por armazenar os dados da imagem que será utilizada para aplicar o filtro.

Para o cálculo da memória do filtro (kernel), a entrada ”endkernel” equivale ao sinal i do contador mencionado anteriormente, e para o cálculo da memória mem (imagem em que

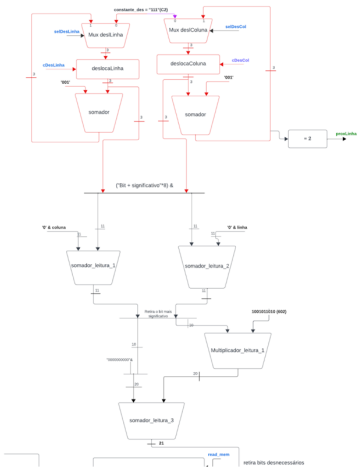

Fig. 4. Diagrama de blocos para o cálculo de endereço da memória mem
Fonte: autoria própria

o filtro será aplicado), a lógica foi que: como o kernel é uma matriz 3x3, temos 3 colunas (coluna -1, 0 e 1), sendo assim, temos um somador que acrescenta +1 até chegar a 2, enviando um sinal para pularmos para a próxima linha do kernel. Os bits armazenados no registrador de deslocamento também são enviados para outra parte do circuito, que implementa uma lógica de soma para calcular o próximo pixel (coluna seguinte e mesma linha, ou seja, o pixel do lado direito) foi adicionada. Como as matrizes foram convertidas em uma única coluna (uma coluna da matriz ficou abaixo da outra) , um multiplicador foi acrescentado, para que seja possível acessar o próximo pixel da mesma linha, que se encontra nos 602 pixels seguintes. Com os endereços calculados, os dados são enviados para a operação de convolução, que consiste em multiplicar os pixels da imagem com os coeficientes da matriz do kernel Após a multiplicação, o kernel exige uma divisão por 16, ou seja, remoção de 4 bits e em seguida, entramos em um loop no somador para finalizar a convolução daquele pixel. Dentro desse loop, temos um multiplexador, em que uma das entradas é utilizada para inserir um valor padrão inicial no registrador e a outra entrada é utilizada para enviar a soma acumulada para o registrador. Ao final, o resultado é armazenado em ”memsaída”, que possui endereço de memória calculado de forma semelhante a mem.

2) ***Bloco de controle****:* O bloco de controle foi construído através de uma máquina de estados finita, com 8 estados que inicia lendo a primeira linha das matrizes de memória. No segundo estado, o sinal para ler a coluna é habilitado juntamente aos sinais de controle de deslocamento. No terceiro estado, os sinais são armazenados nos registradores que enviam para as memórias no estado seguinte. Com os dados em memória, o 5° estado é responsável por realizar toda a operação de multiplicação do pixel, sendo que a operação é acumulado no registrador soma (habilitado neste mesmo estados. No estado seguinte, os sinais de enable dos registradores responsáveis pelo controle do kernel são habitados novamente para que (e

continuam sendo habilitados, até que que "fim i" seja positivo)

3) ***Testbench****:* Dentro do testbench, as três memórias do sistema foram inicializadas: memória de entrada, kernel e memória de saída. Nesse contexto, também foi inicializada a memória do golden model. É relevante ressaltar que, com exceção da memória de saída, todas as outras foram inicializadas pelo arquivo de texto gerado com Python, que produziu um arquivo txt com a matriz em forma de coluna. Para assegurar a confiabilidade do teste, um golden model específico foi desenvolvido. Esse modelo utilizava uma lógica interna para ler o arquivo de texto contendo a matriz da imagem gerada pelo script Python. Posteriormente, o modelo foi integrado ao testbench, permitindo a comparação direta entre a saída gerada pelo circuito e os dados armazenados no golden model. Durante o processamento, o sistema realizou operações conforme programado e armazenou os resultados na memória de saída. Ao final desse processo, uma etapa crucial de verificação ocorreu. Cada valor na memória de saída, representando o resultado do processamento, foi comparado com a correspondente posição na memória do golden model.

III. Resultados

Através do circuito pronto, a imagem 5 foi utilizada para realizar a aplicação do kernel. Os resultados obtidos podem ser visualizados na imagem 6.

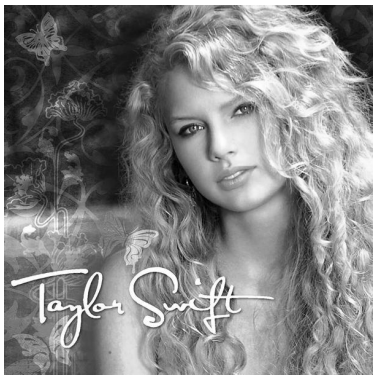

Fig. 5. Imagem modelo antes de aplicar o filtro gaussiano 3x3.
Fonte: Universal Music

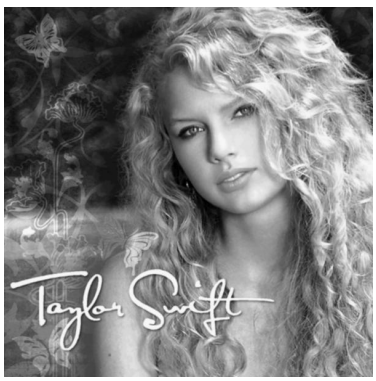

Fig. 6. Resultado após aplicação do filtro gaussiano 3x3

O resultado obtido é melhor visualizado quando aplicamos um zoom (demonstração nas imagens 7 e 8), ou quando o kernel é aplicado mais vezes. Contudo, apesar de sutil, os resultados obtidos foram bastante satisfatórios. Dentro do circuito, um empecilho encontrado foi o tempo de execução da simulação, que para algumas máquinas chegou a horas.

Fig. 7. Recorte da imagem original

Fig. 8. Recorte da imagem após aplicação do filtro

Referências

[1] Wikipedia contributors. (2023, October 17). Kernel (image processing). In Wikipedia, The Free Encyclopedia. Retrieved 01:51, November 29, 2023

[2] Lucidchart criado pelo grupo com os diagramas do projeto