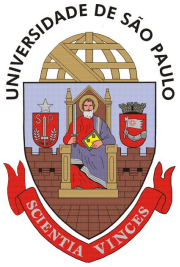

UNIVERSIDADE DE SÃO PAULO

SSC0740

Sistemas Embarcados

Filtro Passa-Alta

Alunos:

Fernando Akio Tutume de Salles Pucci (8957197)
Lucas Nobuyuki Takahashi (10295670)
Vitor Kodhi Teruya (10284441)

São Carlos, 23 de Novembro de 2020

1. Introdução

Neste trabalho é implementado um hardware para aplicar o filtro Passa-alta de duas formas, uma forma tentando diminuir os recursos em *hardware* e outra que maximiza o *throughput*.

O trabalho foi implementado na plataforma EDA playground, que apresenta algumas limitações, como limite de tamanho por arquivo e não permitir leitura de arquivo binário.

1.1. Filtro Passa-Alta

Em processamento de imagens, o princípio da filtragem Passa-alta é realçar as bordas e detalhes de uma imagem, ao contrário do filtro Passa-baixa que mescla os detalhes e relaxa os ruídos. Para atingir esse resultado podemos aplicar uma máscara cuja soma algébrica dos coeficientes é 0 no seguinte padrão:

$$\frac{1}{9}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Figura 1. Máscara de aplicação em um kernel da imagem.

Analisando a máscara, podemos perceber que quando os valores da região são muito parecidos, o *output* é muito próximo a zero. Por outro lado, quando o centro possui valor discrepante, o processamento retorna um valor alto, refletindo assim o comportamento do filtro Passa-alta sobre uma imagem. Um exemplo de output é apresentado abaixo:

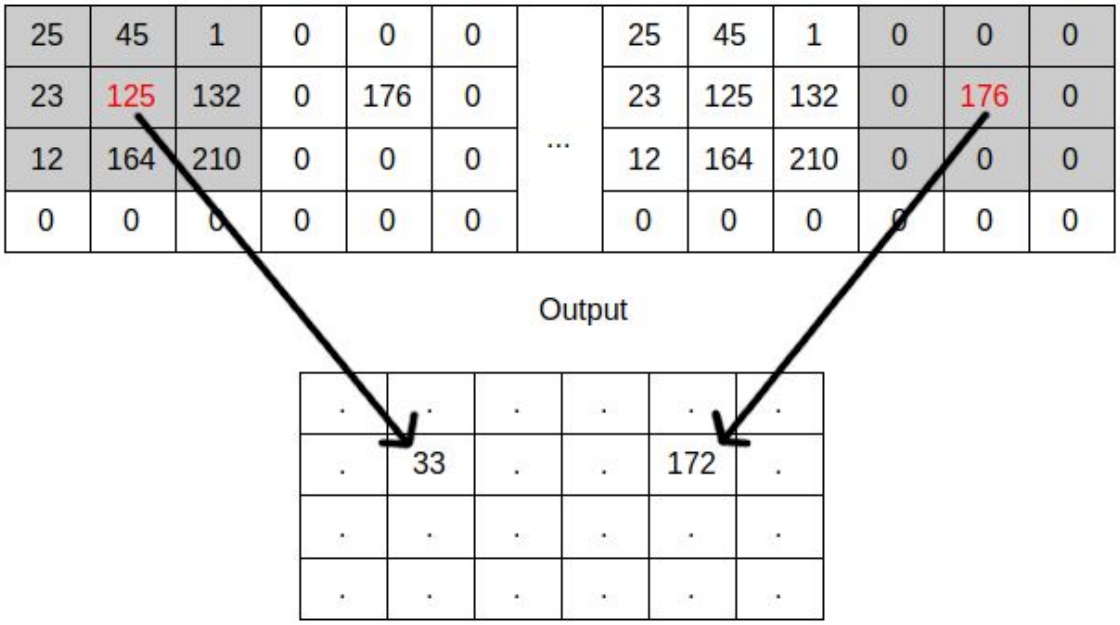

Figura 2. Passa a passo do processamento da entrada.

Para valores no intervalo [0, 255], a saída possui máximo e mínimo de ±2040. A fim de comportar a saída em um intervalo de 1 *byte* o seguinte pós-processamento foi feito:

- saida <= |saida|
- saida <= (saida*255)/2040

Um exemplo do processamento é apresentado abaixo:

Figura 3. Entrada e saída do filtro Passa-alta

2. Implementação

Devido às limitações da plataforma, as imagens utilizadas para o trabalho estão em escala de cinza. Elas foram salvas em 4 arquivos *.txt* onde cada linha da imagem contém um *pixel* da imagem.

Para fazer a divisão dos arquivos, foi utilizado um programa em C que lê um arquivo no formato *.pgm* e salva ela em 4 arquivos que são usados no EDA.

2.1 Minimização de recursos

```
module kernel (
  input [23:0] linha0,
  input [23:0] linha1,
  input [23:0] linha2,
  input clk,
  input start,
  output reg write,
  output reg [7:0] saida
);
```

Para minimização dos recursos, foi feito um *hardware* que faz somente o cálculo do resultado de um *pixel*. Ele recebe 9 *pixels*, que estão divididos pelos *inputs linha0, linha1, linha2*. Além disso o hardware tem uma entrada *start* para indicar quando começar a realizar o cálculo, e o *clk* como *clock*. Como *output,* ele tem o *write* para indicar quando o resultado está pronto, e *saida* como o valor do *pixel* que estava sendo calculado.

```
wire [11:0] fio_kernel;
wire [11:0] fio_kernel_pos;

assign fio_kernel = ( 8 * linha1[15:8]  - linha1[7:0] - linha1[23:16] -
                      linha0[15:8]  - linha0[7:0] - linha0[23:16] -
                      linha2[15:8]  - linha2[7:0] - linha2[23:16]);
assign fio_kernel_pos = fio_kernel[11] == 1'b1 ? -fio_kernel : fio_kernel;

always @(negedge clk) begin
  if(start) begin
    saida <= (fio_kernel_pos * 255) / 2040;
    write <= 1;
  end else begin
    write <= 0;
  end
end
```

O cálculo usado foi o apresentado anteriormente.

Para não ficar muito grande, na linha que faz o cálculo, foram utilizados fios para “armazenar” parte do resultado. O fio *fio_kernel* contém o resultado da multiplicação e subtração dos *pixels* vizinhos.

O fio *fio_kernel_pos* faz o módulo do *fio_kernel*. E por fim o *always* contém a normalização que é salva no *reg saida*.

2.2. Maximização do *Throughput*

Para maximização do *Throughput* foi feito um *pipeline* onde, para cada linha que ele recebe, é feito o cálculo do filtro para uma linha. Para isso o módulo armazena sempre as duas entradas anteriores e a atual, para então fazer o cálculo de cada pixel da linha. Porém, dessa forma, ele calcula sempre o resultado da linha anterior a que foi passada para o módulo. Essa abordagem foi escolhida pensando que mesmo que o hardware consiga fazer o cálculo da imagem inteira de uma vez, o gargalo da transferência da imagem poderia ser o fator limitante e não conseguir alcançar a velocidade máxima do hardware. Pensando assim, o uso dessa abordagem teria um throughput similar ao tempo de leitura da imagem, ou seja, ao mesmo tempo que o *hardware* faz a leitura da imagem, ele faz o cálculo dos *pixels*.

```
module controlador #( parameter N_COLUMNS = 3 )
  ( input [ N_COLUMNS * 8 - 1 : 0 ] entrada,
    input start,
    input reset,
    input finish,
    input clk,
   output write,
   output [ N_COLUMNS * 8 - 1 : 0 ] saida);
```

O módulo tem um parâmetro *N_COLUMNS*, que indica quantos *pixels* tem a linha, que deve ser especificado na instanciação para que o módulo tenha o tamanho de colunas certas.

Como entrada, ele tem um fio *start*, que indica que o módulo pode começar a executar, *reset* que informa que o módulo deva ser *resetado*, *finish* que indica que a imagem que estava sendo passada terminou, *clk* que é o *clock* do módulo, e por último, a entrada, que é a linha da imagem.

Como saída, ele tem o fio *write* que indica quando o módulo começa a liberar resultados válidos, e a saída que é o resultado do filtro para uma linha.

```
reg [ (N_COLUMNS+2) * 8 - 1 : 0 ] linha0 = {(N_COLUMNS+2){8'b0}};
reg [ (N_COLUMNS+2) * 8 - 1 : 0 ] linha1 = {(N_COLUMNS+2){8'b0}};
reg [ (N_COLUMNS+2) * 8 - 1 : 0 ] linha2 = {(N_COLUMNS+2){8'b0}};

wire start_processo;
reg [1:0] contador_start = 2'b0;
reg [2:0] contador_write = 3'b0;

assign start_processo = start & contador_start[1];
assign write = (start & contador_write[0] & contador_write[1]) |
  ( finish & !(!contador_write[0] & !contador_write[1]));

genvar i;
```

Para funcionar, dentro do módulo são instanciados registradores, que representam a *linha0*, linha que acabou de chegar pela entrada. A *linha1*, que é a linha que entrou no ciclo anterior, e é a linha que vai ser calculada. E por fim a *linha2,* que é a linha que foi passada no penúltimo ciclo. Todos esses registradores começam com valor zero. Todas as linhas contém dois *pixels* a mais que são usados como borda, eles sempre serão zeros.

O fio *start_processo* é usado para informar que duas linhas da imagem já estão armazenadas, e que pode começar a realizar o cálculo do filtro. Para poder saber quando ele pode realizar o cálculo, foi feito um *reg contador_start* de 2 *bits* que funciona como um contador para saber quando ele tem 3 linhas. O *reg contador_write* tem o mesmo intuito porém para o fio *write.*

```
genvar i;

generate
  for ( i = 0 ; i < N_COLUMNS ; i = i + 1 ) begin
    kernel h0 ( .linha0(linha0[i * 8 +:24]) ,
                .linha1(linha1[i * 8 +:24]) ,
                .linha2(linha2[i * 8 +:24]) ,
                .clk(clk),
                .start(start_processo),
               .saida( saida[ i * 8 +:8 ]));
    end
endgenerate
```

Para fazer o cálculo do filtro, foi aproveitado o módulo utilizado na minimização do uso de *hardware*. Ele é replicado N vezes, sendo N igual ao número de *pixels* por linha.

```
always @ ( negedge clk ) begin
  if(finish ) begin
    contador_write = contador_write - 1;
  end
  if (reset) begin
    linha0 = {(N_COLUMNS+2){8'b0}};
    linha1 = {(N_COLUMNS+2){8'b0}};
    linha2 = {(N_COLUMNS+2){8'b0}};
    contador_start = 2'b0;
    contador_write = 3'b0;
  end
  if ( !reset & start ) begin
    linha2 <= linha1;
    linha1 <= linha0;
    linha0[(N_COLUMNS+1) * 8 -1 : 8 ] <= entrada & {N_COLUMNS * 8{!finish}};
    if ( !start_processo ) begin
      contador_start = contador_start + 1;
    end
    if ( !write ) begin
      contador_write = contador_write + 1;
    end
  end
end
```

O cálculo todo ocorre dentro do *always*, que contém a lógica para passar as linhas e contém a lógica dos contadores para ativar os fios que funcionam como flags. Ele também contém a lógica para adicionar uma linha de zeros como borda ao final da imagem.

2.3. Métricas de desempenho

Para a extração das métricas de desempenho e *hardware*, como o *Throughput*, *Timing* e *Latency,* houve a tentativa de utilização do Simulador *Mentor Precision* dentro do ambiente *EDA playground.* Mas o simulador apresentou erros durante a fase de compilação e não foi possível diagnosticar o erro pela falta de *log* que a plataforma oferece. Mesmo após ler a documentação do simulador e estudar o documento de configuração *run.do,* o erro persistiu. Dessa forma os cálculos foram feitos baseados em quantidade de ciclos.

Para o hardware que tenta diminuir o uso de hardware (considerando que a imagem já está toda em memória), ele levaria a quantidade de pixels da imagem. Com isso em mente, a latência do *hardware* seria de 76800 ciclos de clock e teria um *throughput* de uma imagem a cada 76800 ciclos.

O hardware com *throughput* maximizado levaria a quantidade de linhas que a imagem tem + 3 (N linhas + 3) ciclos de *clock.* Dessa forma ele teria latência de 243 ciclos de clock, e teria um *throughput* de 1/243 imagens por ciclo de clock.

3. Conclusão

Durante a implementação do algoritmo Passa-alta, ficou claro que a replicação dos módulos de processamento aumentaria a taxa de *throughput* do sistema, de uma maneira análoga aos fluxos de cálculos que acontecem nas *GPU*. Por outro lado, os recursos em *hardware* aumentam bastante com a replicação dessas unidades. A utilização de um *pipeline* é a melhor opção para tarefas no contexto de processamento de imagens, mas é necessário realizar uma análise para otimizar o paralelismo, levando em conta o ganho de desempenho em prol dos recursos.