Projeto final -

INF01046 - FUNDAMENTOS DE PROCESSAMENTO DE IMAGENS

Matheus Leite Cruz

10 de Maio de 2021

Sumário

1 Introdução

O desenvolvimento de aplicações eficientes de processamento de imagens é complexo: Além da produzir um resultado correto, é necessário que um algorítimo performático leve em consideração diversas características do sistema operacional e hardware (localidade de cache, paginação, múltiplos processadores) em sua implementação para obter o melhor resultado possível. Sendo assim, implementações otimizadas e paralelas são extremamente complexas, com alto custo de manutenção e baixa legibilidade.

O artigo *Decoupling Algorithms from Schedulesfor Easy Optimization of Image Processing Pipelines*[1] argumenta que essa complexidade é causada pela falta de separação explicita entre *algorítimo* e *escalonamento*(scheduling), e propõe uma uma representação que permite expressar algoritmos separados de sua estrategia de execução.

A representação apresentada, nomeada de *Halide* foi implementada como uma DSL para a linguagem c++. O objetivo desse relatório é apresentar as características da representação, explicar suas motivações e demonstrar algumas implementações de algorítimos de processamento de imagem utilizando Halide.

2 Implementando programas em Halide

A linguagem Halide foi implementada em c++ utilizando características como overloading de operadores, se tratando de c++ válido.

Programas em Halide apresentam um estilo funcional: Algorítimos são implementados como funções puras operando sobre conjuntos de números, completamente alheios a questões de implementação, e estratégias de execução podem ser definidas separadamente para cada algorítimo. O programa é então compilada para uma representação intermediaria otimizada que implementa de maneira imperativa o algoritmo em questão utilizando sua estratégia de execução.

2.1 Definindo um algoritmo

2.1.1 Expressões, Funções, Variáveis

Uma função (Func) pode representar um estagio em uma pipeline de processamento de imagens. Cada função possui um número qualquer de variáveis (Var) e cada variável de uma função representa o conjunto dos números inteiros. Funções são definidas em termos de expressões (Expr) que utilizam suas variáveis e outras funções para computar um valor final em alguma (ou em um conjunto de) coordenadas. Por exemplo, uma função que utiliza as variáveis X, pode ser vista como uma função operando sobre os eixos X e Y do plano cartesiano.

Funções operam sobre domínios infinitos e não tem *side effects*, então sua execução só acontece combinada com um plano de execução e um intervalo de valores.

Por exemplo, o código abaixo calcula x + y para todos os pixels(x,y) de uma matriz 800x600:

```
Halide::Func gradiente;
Halide::Var x,y;

// x + y é uma Expr
gradiente(x,y) = x + y

// 800, 600 limitam x e y
// realize executa o algoritmo com a estretegia padrão
Buffer<u_int8_t> resultado = gradiente(800,600).realize()
```

2.1.2 Updates

Funções podem utilizar valores computados chamadas anteriores para computar novos valores (Update):

```
Halide::Func gradiente;
Halide::Var x,y;

gradiente(x,y) = x + y
// Update: Zera primeira coluna
gradiente(x,0) = 0
```

2.1.3 Reduções

Reduções representam um 'loop' realizando updates:

```
// considere image como já definido
// image em tom de cinza
Halide::Func histograma;
Halide::Var h;

// Dominio de redução 2D:
// X variando de 0 até width
// Y viariando de 0 até height
r = Rdom(0, image.width, 0, image.height);

// Inicializa com 0
histograma(h) = 0.0f;

// Para cada x,y em r correspondente a um pixel da imagem
// é somado +1 no indice do histograma correspondente
histograma(image(r.x,r.y)) += 1;

//Equivalente à:
```

```
for(int i=0;i < image.width; i++){
    for(int j=0; j< image.height){
        histograma(image(i,j)) += 1;
    }
}
```

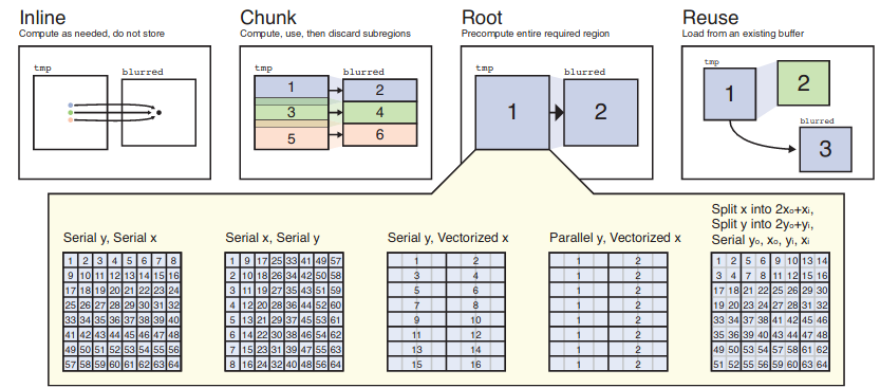
2.2 Definindo uma estratégia de execução

Após definido o algoritmo, é possível definir sua estratégia de execução separadamente. Para os exemplos de estratégia de execução, utilizaremos a seguinte função:

```
Func gradient("gradiente");
gradient(x, y) = x + y;
```

A estrategia de execução padrão computa os valores de cada função conforme forem necessários (inline) e itera sequencialmente sobre os valores (serial). É possível combinar diversas estratégias de execução para atingir melhores resultados, ou até mesmo gerar automaticamente estratégias eficientes. Apresentamos alguns poucos exemplos a seguir, mas é bom notar que é possível realizar diversas outras estrategias além das apresentadas

Figura 1: Exemplos de estrategia de execução. Fig.4 - *Decoupling Algorithms from Schedulesfor Easy Optimization of Image Processing Pipelines*



2.2.1 Compute root

A estratégia compute root realiza o calculo de todos os valores da função e salva no buffer. Diferentemente da estrategia padrão(inline), a função é computada somente uma vez (trocando tempo de execução por uso de memoria).

```
gradient.compute_root();
```

2.2.2 Reorder

Reordena loops sobre variáveis. Por exemplo a estrategia serial padrão utiliza como loop mais externo a ultima variável da função. A função reorder possibilita reordenar os loops. Utilizamos a função print_loop_nest para observar a ordem dos loops.

```
//Padrão:
// > produce gradient:
// >   for y:
// >     for x:
// >       gradient(...) = ...
gradient.print_loop_nest();
gradient.reorder(y,x);
// Resulta em:
// > produce gradient:
// >   for x:
// >     for y:
// >       gradient(...) = ...
```

2.2.3 Vectorize

Separa loops em loops internos e externos, utilizando operações vetoriais para processar o loop interno. Um exemplo de uso dessa estratégia é ter um loop interno que possa ser processado de uma só vez utilizando uma única operação vetorial.

```
gradient.print_loop_nest();

gradient.vectorize(x,4);
// Resulta em:
// produce gradient:
//  >   for y:
//  >     for x.x_outer:
//  >       vectorized x.x_inner in [0, 3]:
//  >         gradient(...) = ...
```

3 Implementações

Foram implementados alguns algorítimos em Halide para demonstrar a linguagem. Os algoritmos escolhidos apresentam níveis variáveis de complexidade.

3.1 Alterar Brilho

Em Halide, o algorítimo de alteração de brilho de uma imagem é trivial:

```
// Converte e faz clamping do valor
Expr clamp(Expr expr)
{
    return Halide::cast<u_int8_t>(Halide::min(255, expr));
}

void adjust_brightness(Buffer<u_int8_t> src, int val){

    // C varia de 0 a 2 e representa os canais
    // BGR de cor da imagem
    Var x,y,c;
    Func adjust_brightness;
    adjust_brightness(x,y,c) = clamp(src(x,y,c + val);
    auto output = adjust_brightness.realize(src.width(),src.height(),3);

}
```

3.2 Tons de cinza

Em Halide, o algorítimo de conversão para cinza é trivial:

```
// Converte e faz clamping do valor
Expr clamp(Expr expr)
{
    return Halide::cast<u_int8_t>(Halide::min(255, expr));
}

void to_grayscale(Buffer<u_int8_t> src, int val){

    // C varia de 0 a 2 e representa os canais
    // BGR de cor da imagem
    Var x,y;
    Func to_grayscale;
    to_grayscale(x,y) = clamp(buf(x, y, 0) * 0.114f + buf(x, y, 1) * 0.587f + buf(x, y, 2) *
    auto output = to_grayscale.realize(src.width(),src.height(),3);

}
```

3.3 Filtro 3x3

O algoritmo para um filtro 3x3 é razoavelmente simples. É interessante notar que funções para facilitar o tratamento de condições de borda são disponibilizadas pela linguagem, simplificando o trabalho de implementação.

```
// Converte e faz clamping do valor
Expr clamp(Expr expr)
{
```

```
    return Halide::cast<u_int8_t>(Halide::min(255, expr));
}

void filter_3x3(Buffer<u_int8_t> src){
    Var x("x"), y("y"), c("c");
    Func filter_3x3("filter_3x3");
    Func src_int;

    // Tratamento de bordas
    Func clamped = BoundaryConditions::repeat_edge(src);

    // Transformamos para inteiro para não ter overflow
    src_int(x, y, c) = cast<int>(clamped(x, y, c));

    // Kernel 3x3
    // k0 k1 k2                    (-1,-1) (0,-1)  (1,-1)
    // k3 k4 k5  Aplicado em =>    (-1, 0) (0, 0)  (1, 0)
    // k6 k7 k8  relativo a origem (-1, 1) (0, 1)  (1, 1)

    int k0 = 1, k1 = 1, k2 = 1, k3 = 1, k4 = 1, k5 = 1, k6 = 1, k7 = 1, k8 = 1;
    int w = (k0 + k1 + k2 + k3 + k4 + k5 + k6 + k7 + k8);

    Expr expr = src_int(x - 1, y - 1, c) * k0
                + src_int(x, y - 1, c) * k1
                + src_int(x + 1, y - 1, c) * k2
                + src_int(x - 1, y, c) * k3
                + src_int(x, y, c) * k4
                + src_int(x + 1, y, c) * k5 +
                + src_int(x - 1, y + 1, c) * k6
                + src_int(x, y + 1, c) * k7
                + src_int(x + 1, y + 1, c) * k8;

    // utilizamos a expressão divida pelo peso
    // para calcular o resultado
    filter_3x3(x, y, c) = clamp(expr / w);
    auto output = filter_3_3.realize(src.width(),src.height(),3);
}
```

3.4 Equalização de histograma

O algorítimo de equalização de histograma é um pouco mais complicado, por precisar de vários passos e reduções:

```
// Converte e faz clamping do valor
Expr clamp(Expr expr)
{
```

```
    return Halide::cast<u_int8_t>(Halide::min(255, expr));
}

// Histograma não normalizado da imagem tons de cinza
Func histogram(Buffer<u_int8_t> src)
{
    Var x("x"), y("y"), c("c");
    Var h("h");

    Func grayscale("grayscale");
    Func histogram("histogram");

    // Tons de cinza
    grayscale(x, y, c) = to_grayscale(src, x, y);
    grayscale.compute_root();

    // reduction domain variando em 2d
    auto x_y_domain = RDom(0, src.width(), 0, src.height(), "x_y");

    // Histograma da imagem
        histogram(h) = 0.0f;
        histogram(grayscale(x_y_domain.x, x_y_domain.y, 0)) += 1.0f;
    return histogram;
}

void histogram_equalization(Buffer<u_int8_t> src)
{
    Var x("x"), y("y"), c("c");
    Var h("h");

    float alpha = 255.0f / (src.height() * src.width() / 3);

    // Utilizamos a função histograma anterior
    Func histogram_fn = histogram(src);
    histogram_fn.compute_root();

    Func hist_cum("hist cum");

    hist_cum(h) = alpha * histogram_fn(h);

    // Redução de 1 até 255
    auto h_dom = RDom(1, 255, "h");

    // Calculo do histograma cumulativo
    hist_cum(h_dom.x) =
        hist_cum(h_dom.x - 1) +
```

```
        ((alpha * histogram_fn(h_dom.x)) / 3);

    hist_cum.compute_root();;

    Func histogram_eq("histogram eq");
    histogram_eq(x, y, c) = clamp(hist_cum(src(x, y, c)));

    auto output = histogram_eq.realize(800, 600, 3);
}
```

Exemplo de loop gerado pela função histogram_eq() para a imagem de teste Gramado_72k:

```
produce histogram:
  for h in [0, 255]:
    histogram(...) = ...
  for x_y in [0, 239]:
    for x_y in [0, 319]:
      histogram(...) = ...
consume histogram:
  produce hist cum:
    for h in [0, 255]:
      hist cum(...) = ...
    for h in [1, 255]:
      hist cum(...) = ...
  consume hist cum:
    produce histogram eq:
      for c:
        for y:
          for x:
            histogram eq(...) = ...
```

4 Resultados

A partir dos algoritmos implementados, é extremamente simples iterar interativamente sobre possíveis estrategias de execução para cada um. Por exemplo, partindo da função de equalização de histograma é possível implementar facilmente estratégias de execução extremamente complexas sem alterar o algoritmo.

Por exemplo, implementamos a seguinte estratégia para a função histogram_eq:

1. Dividir a imagem em blocos de 64x64
2. Iterar de maneira paralela sobre os blocos
3. Para cada bloco, divide o bloco em blocos de 4x4

4. Cada sub-bloco é formado por um loop externo e um interno (4 iterações)
5. Cada iteração externa do sub-bloco desenrolada
6. Cada iteração interna do sub-bloco é vetorizada

Implementação da estratégia de execução:

```
// Inspiração para estratégia:
// https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

Var x_outer("x_outer"), y_outer("y_outer");
Var x_inner("x_inner"), y_inner("y_inner");
Var tile_index("tile_index");

// Processamos parelelamente em blocos de 64x64
histogram_eq.tile(x, y, x_outer, y_outer, x_inner, y_inner, 64, 64)
    .fuse(x_outer, y_outer, tile_index)
    .parallel(tile_index);

// Separamos cada bloco de 64 em blocos de 4x4
// Vetorizando o loop externo tamanho 4
// e realizando unroll no loop interno tamanho 4
Var x_inner_outer("x_inner_outer"), y_inner_outer("y_inner_outer"), x_vectors("x_vectorts"),
histogram_eq
    .tile(x_inner, y_inner, x_inner_outer, y_inner_outer, x_vectors, y_pairs, 4, 4)
    .vectorize(x_vectors)
    .unroll(y_pairs);

histogram_eq.print_loop_nest();
```

Loop gerado pela estratégia de execução acima:

```
produce histogram:
  for h in [0, 255]:
    histogram(...) = ...
  for x_y in [0, 239]:
    for x_y in [0, 319]:
      histogram(...) = ...
consume histogram:
  produce hist cum:
    for h in [0, 255]:
      hist cum(...) = ...
    for h in [1, 255]:
      hist cum(...) = ...
  consume hist cum:
    produce histogram eq:
      for c:
```

```
parallel x.x_outer.tile_index:
  for y.y_inner.y_inner_outer in [0, 15]:
    for x.x_inner.x_inner_outer in [0, 15]:
      unrolled y.y_inner.y_pairs in [0, 3]:
        vectorized x.x_inner.x_vectorts in [0, 3]:
          histogram eq(...) = ...
```

Logo é possível notar que a separação do algoritmo da estratégia execução resulta em implementações mais simples e legíveis. Além disso, é extremamente fácil testar diversas estrategias de execução e implementar programas que seriam extremamente complexos de serem implementados de qualquer outra forma.

Infelizmente dado os limites de tempo não foi possível implementar nenhum algoritmo que operasse no domínio de frequência (o tratamento de números complexos é razoavelmente complicado). Também não foi possível utilizar ferramentas de medição de performance (perf e outros) para comparar o impacto real no sistema de estratégias de execução diferentes.

Os arquivos do projeto podem ser encontrados online em : ⟨https://github.com/mlcruz/fpi-halide⟩

5 Conclusão

O artigo *Decoupling Algorithms from Schedulesfor Easy Optimization of Image Processing Pipelines* identifica corretamente problemas causados pelo acoplamento da estratégia de execução ao algoritmo e com sucesso apresenta uma solução que resolve parte do problema.

A linguagem Halide consegue expressar de maneira natural diversos algorítimos de processamento de imagem, separando completamente o algoritmo da execução de maneira impressionantemente eficiente. É incrível como diversos algoritmos complexos podem ser implementados de maneira simples e eficiente utilizando a linguagem.

Além disso a possibilidade de implementar estrategias de execução genéricas em relação aos algorítimos permite com que seja simples reaproveitar ou até mesmo gerar estratégias automaticamente.

Referências

1 RAGAN-KELLEY, J. et al. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, Association for Computing Machinery, New York, NY, USA, v. 31, n. 4, jul. 2012. ISSN 0730-0301. Disponível em: ⟨https://doi.org/10.1145/2185520.2185528⟩.