Introdução à Halide

Linguagem de Programação Específica para IA - Lux.AI

INSTITUIÇÃO EXECUTORA

Centro de Informática UFPE

FADE UFPE

COORDENADORA

MCTI FUTURO

Softex

APOIO

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÃO

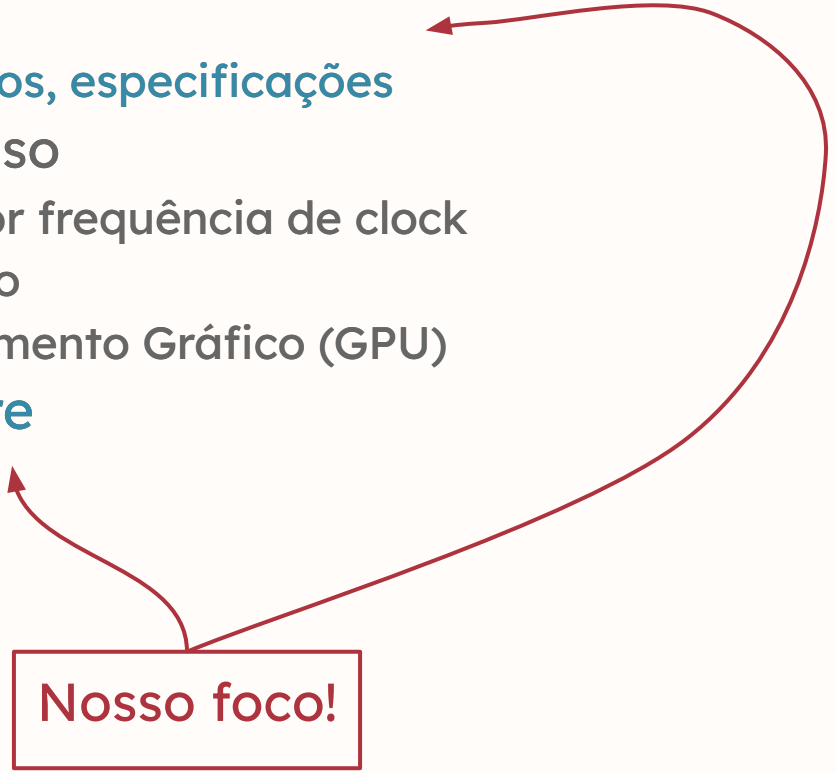
GOVERNO FEDERAL BRASIL UNIÃO E RECONSTRUÇÃO

Eficiência no processamento de imagens

- ➔ Eficiência algorítmica
  - ◆ Algoritmos aproximados, especificações
- ➔ Hardware mais poderoso
  - ◆ Processador com maior frequência de clock
  - ◆ Mais *core*s, vetorização
  - ◆ Unidades de Processamento Gráfico (GPU)
- ➔ Melhor uso do hardware

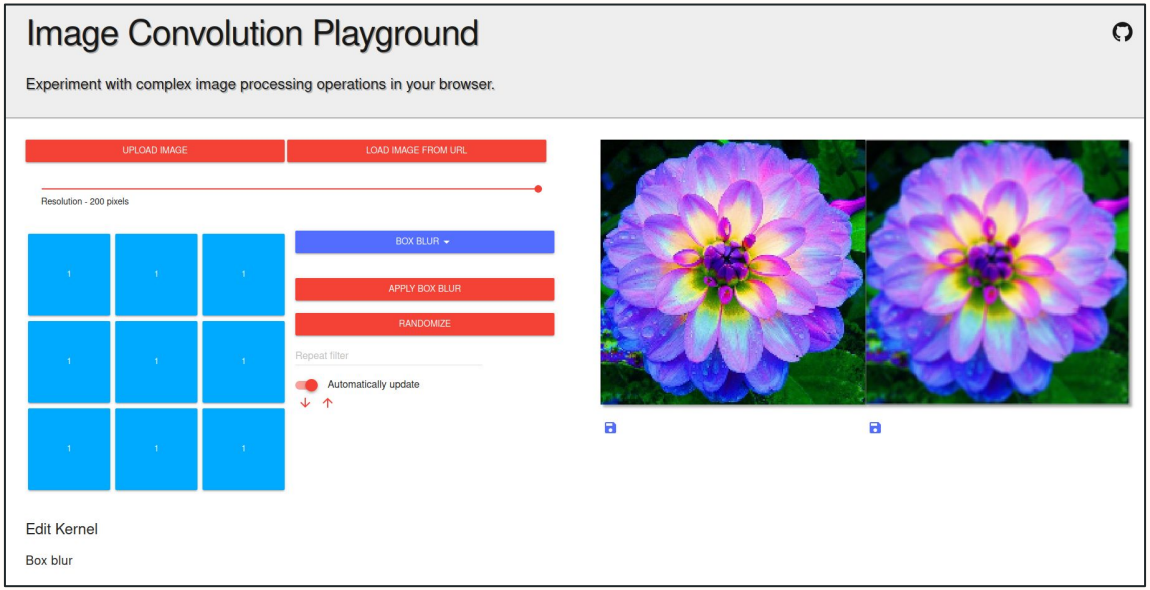
Eficiência no processamento de imagens

- Eficiência algorítmica
  - Algoritmos aproximados, especificações
- Hardware mais poderoso
  - Processador com maior frequência de clock
  - Mais *core*s, vetorização
  - Unidades de Processamento Gráfico (GPU)
- Melhor uso do hardware



Exemplificando

➔ Filtro de caixa 3x3



https://generic-github-user.github.io/Image-Convolution-Playground/src/

Implementação de filtro de caixa 3x3

C++ força bruta

Fonte: DOI:10.1145/3150211

(a) Clean C++: 6.5ms per megapixel

```
void blur(const Image<uint16_t> &in, Image<uint16_t> &bv) {
 Image<uint16_t> bh(in.width(), in.height();

  for (int y = 0; y < in.height(); y++)
   for (int x = 0; x < in.width(); x++)
    bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

  for (int y = 0; y < in.height(); y++)
   for (int x = 0; x < in.width (); x++)
    bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;
}
```

Filtro de caixa 3x3 - C++ simples

(a) Clean C++: 6.5ms per megapixel

```
void blur(const Image<uint16_t> &in, Image<uint16_t> &bv) {
 Image<uint16_t> bh(in.width(), in.height();

  for (int y = 0; y < in.height(); y++)
   for (int x = 0; x < in.width(); x++)
    bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

  for (int y = 0; y < in.height(); y++)
   for (int x = 0; x < in.width (); x++)
    bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;
}
```

Implementação de filtro de caixa 3x3

C++ otimizado à mão

Fonte: DOI:10.1145/3150211

(b) Fast C++ (for x86) : 0.30ms per megapixel

```
void fast_blur(const Image<uint16_t> &in, Image<uint16_t> &bv) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i bh[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *bhPtr = bh;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
       a = _mm_loadu_si128((__m128i*)(inPtr - 1));
       b = _mm_loadu_si128((__m128i*)(inPtr + 1));
       c = _mm_load_si128 ((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(bhPtr++, avg);
     inPtr += 8;
   }}
   bhPtr = bh;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(bv(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
       a = _mm_load_si128(bhPtr + (256 * 2) / 8);
       b = _mm_load_si128(bhPtr + 256 / 8);
       c = _mm_load_si128(bhPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

Contextualização

Dilema Portabilidade-Especificidade

- Portabilidade:
  - A portabilidade do software refere-se a sua capacidade de ser **facilmente transferido** de um ambiente de computação para outro, **mantendo seu desempenho** e funcionalidade.
- Especificidade:
  - Especificidade de hardware refere-se a **características específicas** de um determinado hardware que executará um algoritmo.

Se um código tem alta portabilidade significa que ele não pode explorar recursos específicos de um ambiente de computação, pois esses recursos podem não estar disponíveis em outro ambiente.

“Dilema” código limpo - código eficiente

- Código limpo:
  - se refere à prática de escrever código de maneira clara, legível e fácil de entender para que possa ser facilmente compreendido por outros desenvolvedores
- Código eficiente:
  - código feito com objetivo de aumentar a performance da execução.

Muitas vezes, o **requisito de eficiência** de um determinado sistema necessita de um **código mais específico** para o hardware que será executado. Essas especificações podem gerar um **código de alta complexidade e difícil legibilidade**.

Outras dificuldades na otimização

- ➔ Necessidade de alto nível de conhecimento específico
- ➔ Necessidade de reescrita para diferentes ambientes de execução
- ➔ Muito tempo gasto para explorar otimizações
  - ◆ Modificar a organização da computação geralmente significa alterar a muito do código

- ➔ Nas implementações de pipelines com bibliotecas (ex.: OpenCV)
  - ◆ Os operadores são isoladamente otimizados deixando muitas opções de otimização de pipeline inexploradas

Otimização da Execução

Paralelismo

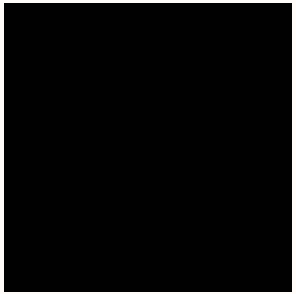
Paralelismo

"Paralelismo" refere-se à **execução simultânea** de várias tarefas ou instruções de um programa.

- Paralelismo de Tarefas
    - Envolve a execução simultânea de **tarefas distintas** em diferentes unidades de processamento. Isso é mais comum em sistemas que executam várias aplicações ou processos ao mesmo tempo, como em sistemas operacionais multitarefas.
- Paralelismo de Dados
    - Envolve a divisão de uma grande tarefa em partes menores que podem ser executadas simultaneamente. **Cada parte opera nos dados independentemente**, utilizando recursos de processamento separados. Isso é comum em operações intensivas de dados, como em **processamento de imagens**.

Paralelismo - exemplos

Mostraremos exemplos relacionados com a criação de um gradiente de tons de cinza como no gif a seguir:

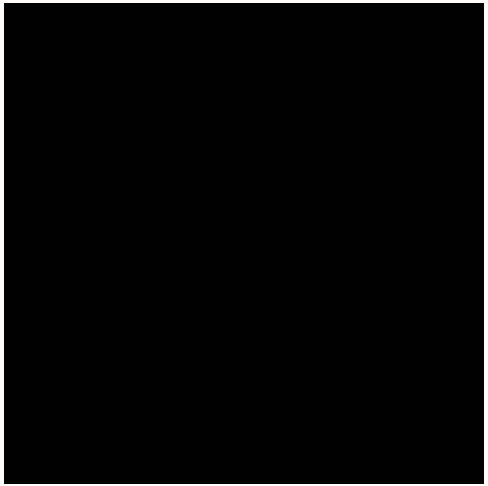


A intensidade de cada pixel é dada pela fórmula: row+col, em que row é a linha do pixel e col é a coluna dele.

Fonte: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

Paralelismo - exemplos

Acesso com paralelismo



Fonte: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

Vetorização

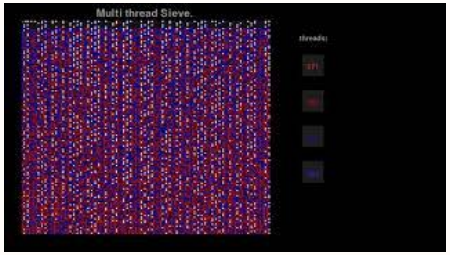
Consiste na execução de uma mesma instrução à múltiplos dados simultaneamente (**S**ingle **I**nstruction **M**ultiple **D**ata, SIMD)

Fonte: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

Paralelismo - exemplos de outras áreas

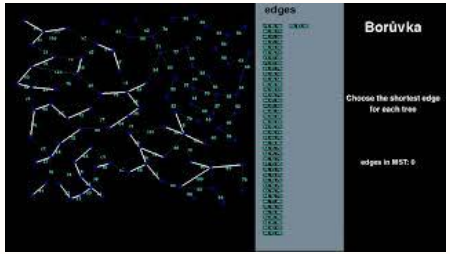
O crivo de Eratóstenes: comparação de algoritmos na busca por números primos

https://youtu.be/SPnIuFn27V0

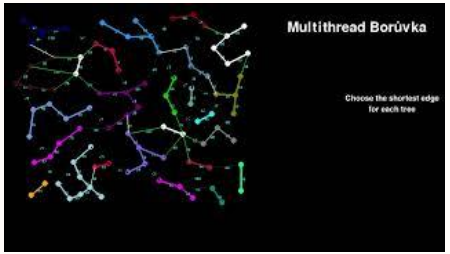


https://youtu.be/fpqGRVtBER4

O algoritmo de Boruvka: *minimum spanning trees*:



https://youtu.be/rOiZOXMViPY



https://youtu.be/6ODPqo1dVrs

Baixa Redundância

Baixa Redundância

Simplesmente consiste em evitar realizar uma operação previamente realizada.

Geralmente isto é feito de 2 formas:

- **organizando a ordem das operações** que são feitas;
- **guardando na memória** os resultados que serão necessários posteriormente.

Exemplo Clássico - Fibonacci

Queremos uma função que nos retorne um número da sequência de Fibonacci , F(n), dado o input inteiro n > -1, considerando F(1) = F(0) = 1.

A fórmula da sequência de Fibonacci é: F(k) = F(k-1) + F(k-2), em que k é inteiro e maior que 1.

$$\mathbf{F(2)} = F(1) + F(0) = \mathbf{2}$$

$$\mathbf{F(3)} = F(2) + F(1) = F(1) + F(0) + F(1) = \mathbf{3}$$

$$\mathbf{F(4)} = F(3) + F(2) = F(1) + F(0) + F(1) + F(1) + F(0) = \mathbf{5}$$

Exemplo Clássico - Fibonacci

Solução com muita redundância

```
int Fibonacci(int n){
    if(n == 0 || n == 1) return 1;
    return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Exemplo Clássico - Fibonacci

Solução com memorização

```
// criando um espaco na memoria
int F_memorizado[N]; // considere N suficiente
F_memorizado[0] = F_memorizado[1] = 1;
for(int i=2; i<N; i++)
    //marcando como ainda nao calculado
    F_memorizado[i] = -1;
```

Exemplo Clássico - Fibonacci

Solução com memorização

```
int Fibonacci(int n){
    // se ja temos o numero de Fibonacci solicitado,
    // retornamos ele sem repetir as operacoes
    if (F_memorizado[n] != -1) return F_memorizado[n];

    //se ainda nao temos o numero de fibonacci da memoria,
    // calculamos
    // guarda os resultados das operacoes na memoria
    F_memorizado[n-1] = Fibonacci(n-1);
    F_memorizado[n-2] = Fibonacci(n-2);

    //retorna o resultado
    return F_memorizado[n-1] + F_memorizado[n-2];
}
```

Exemplo Clássico - Fibonacci

Solução com organização dos cálculos

```
int Fibonacci(int n){
    if(n == 0 || n == 1) return 1;

    int fi_2 = 1; //Fibonacci(i-2)
    int fi_1 = 1; //Fibonacci(i-1)
    int fi = 2;   //Fibonacci(i)

    for(int i = 1; i < n; i++){
        fi = fi_2 + fi_1; //calcula Fibonacci(i)
        // incrementa i
        fi_2 = fi_1;
        fi_1 = fi;
    }
    return fi;
}
```

Evitar Cálculos Desnecessários

```
int Fibonacci(int n) {
    if (n == 0 || n == 1) return 1;
    if (n == 2) return 2;
    if (n == 3) return 3;

    int fi_2 = 1, fi_1 = 2, int fi = 3;

    int i;
    for (i = 2; i < n; i+=2) {
        fi = fi_2 + fi_1; // calcula Fibonacci(i)
        // incrementa i
        fi_2 = fi_1;
        fi_1 = fi;

        fi = fi_2 + fi_1; // calcula Fibonacci(i+1)
        // incrementa i
        fi_2 = fi_1;
        fi_1 = fi;
    }

    if(i <= n) return fi;
    return fi_2;
}
```

Evitar Cálculos Desnecessários

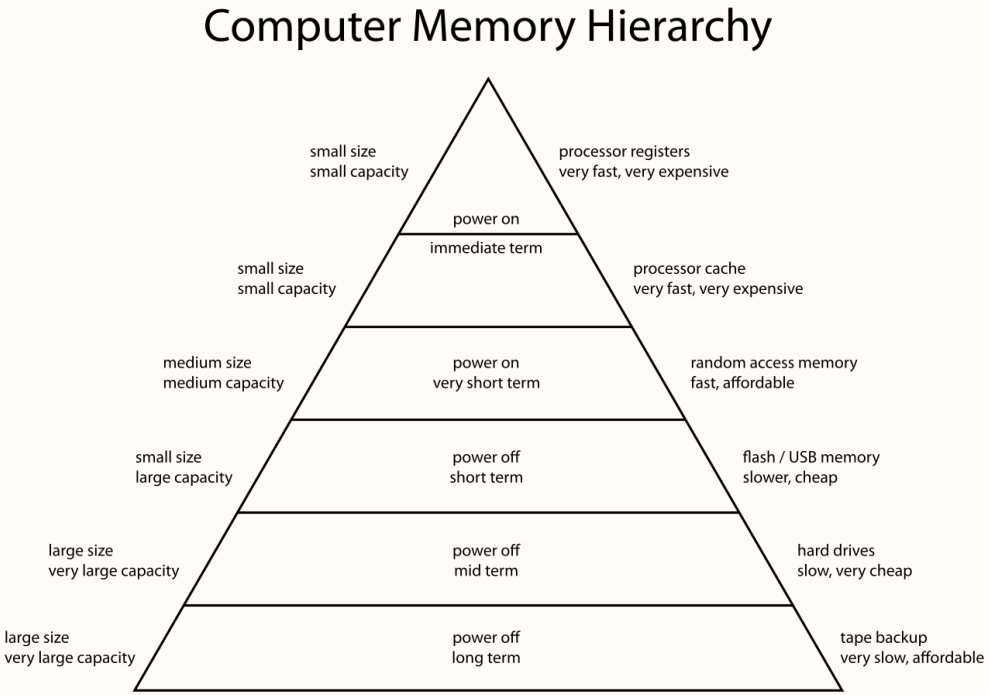
Loop Unrolling

```
for (i = 2; i < n; i+=2) {
    fi = fi_2 + fi_1; // calcula Fibonacci(i)
    // incrementa i
    fi_2 = fi_1;
    fi_1 = fi;

    fi = fi_2 + fi_1; // calcula Fibonacci(i+1)
    // incrementa i
    fi_2 = fi_1;
    fi_1 = fi;
}
```

Localidade

Funcionamento da memória

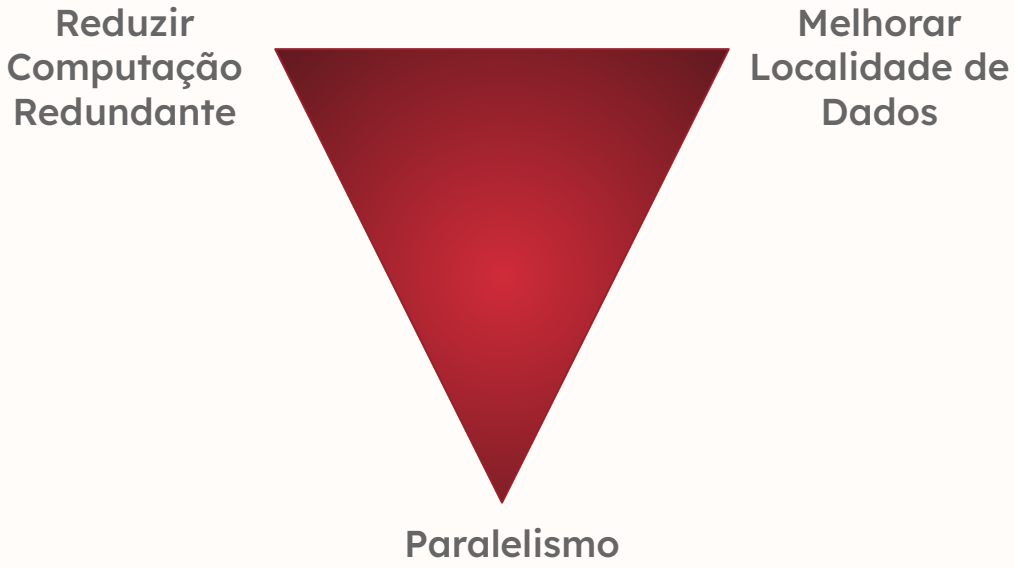


Fonte: https://en.wikipedia.org/wiki/Memory_hierarchy

Princípio de Localidade

- ➔ Programas exibem localidade:
  - ◆ Localidade espacial: Após se acessar um endereço de memória, os endereços vizinhos tem alta probabilidade de serem acessados (ex.: iterar por um array)
  - ◆ Localidade temporal: Após se acessar um endereço de memória, o mesmo endereço tem alta probabilidade de ser acessado novamente (ex.: acesso a um dado usado em um loop)
- ➔ A interação com a memória tende a representar um gargalo nas pipelines. Portanto, explorar os padrões de acesso à memória melhora a localidade e pode trazer aumento imediato na performance do programa

“Trilema” da eficiência



Como Halide Ajuda?

Solução do Halide

Algoritmo

Instruções que indicam o que deve ser calculado.

Independente do ambiente de execução.

Mudanças no algoritmo podem mudar o output para o mesmo input

Schedule

Instruções de como o algoritmo deve ser calculado.

Deve ser ajustado para performar melhor em cada ambiente de execução.

Mudanças no schedule não alteram o output para o mesmo input.

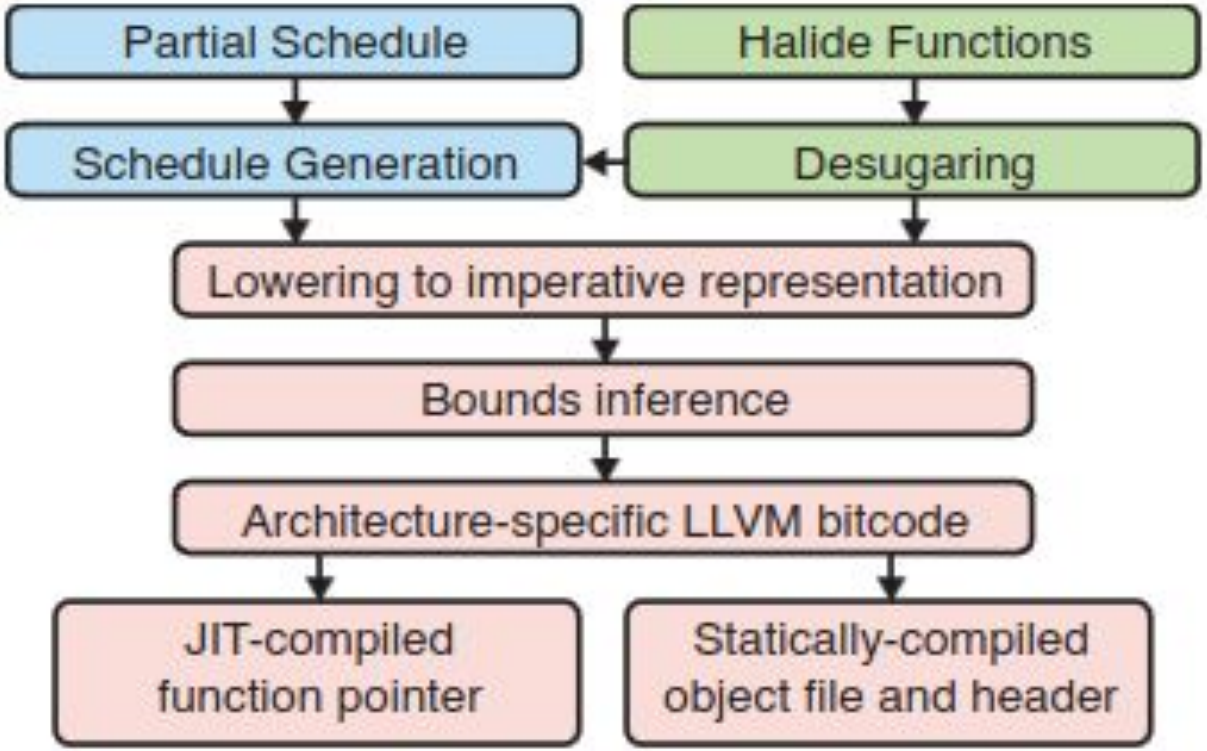
Solução do Halide

- ➔ Halide é uma linguagem de domínio específico **funcional** para algoritmos de processamento de arrays integrada com C++
- ➔ Sua utilização ocorre através da metaprogramação

Solução do Halide

- ➔ A intenção por trás de Halide **não é propriamente fornecer novas técnicas** de explorar maximização de localidade e paralelismo e minimização de redundância.
- ➔ O que Halide fornece é uma **sintaxe que facilita/acelera a experimentação** com técnicas já bem difundidas sem que a corretude do algoritmo seja afetada.
- ➔ Além disso, utilizando diretivas para representar o schedule, Halide oferece código limpo sem sacrificar eficiência e código portável sem sacrificar especificidade
  - ◆ Ambos os aspectos ficam escondidos do desenvolvedor sendo gerenciados pelo sistema de geração de código de Halide
- ➔ É importante ressaltar que Halide não possui suporte interno à algoritmos iterativos nem a recursão sem limites pré-estabelecidos

O compilador de Halide



Fonte: https://people.csail.mit.edu/jrk/halide12/halide12.pdf

Implementação de filtro de caixa 3x3

Halide

Fonte: DOI:10.1145/3150211

(c) Halide : 0.29ms per megapixel

```
Func halide_blur(Func in) {
 Func bh, bv;
 Var x, y, xi, yi;

 // The algorithm
 bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;

 // The schedule
 bv.tile(x, y, xi, yi, 256, 32)
   .vectorize(xi, 8).parallel(y);
 bh.compute_at(bv, x).vectorize(x, 8);

 return bv;
}
```

Filtro de caixa 3x3 - Halide

(c) Halide : 0.29ms per megapixel

```
Func halide_blur(Func in) {
 Func bh, bv;
 Var x, y, xi, yi;

 // The algorithm
 bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;

 // The schedule
 bv.tile(x, y, xi, yi, 256, 32)
   .vectorize(xi, 8).parallel(y);
 bh.compute_at(bv, x).vectorize(x, 8);

 return bv;
}
```

Filtro de caixa 3x3 - Halide

(c) Halide : 0.29ms per megapixel

```
Func halide_blur(Func in) {
 Func bh, bv;
 Var x, y, xi, yi;

 // The algorithm
 bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;

 // The schedule
 bv.tile(x, y, xi, yi, 256, 32)
   .vectorize(xi, 8).parallel(y);
 bh.compute_at(bv, x).vectorize(x, 8);

 return bv;
}
```

Algoritmo

Filtro de caixa 3x3 - Halide

(c) Halide : 0.29ms per megapixel

```
Func halide_blur(Func in) {
 Func bh, bv;
 Var x, y, xi, yi;

 // The algorithm
 bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;

 // The schedule
 bv.tile(x, y, xi, yi, 256, 32)
   .vectorize(xi, 8).parallel(y);
 bh.compute_at(bv, x).vectorize(x, 8);

 return bv;
}
```

Algoritmo

Schedule

Resumo de prós e contras

#	Clean C++	Fast C++	Halide
Desempenho (box blur)	`6.5ms/MP`	`0.3ms/MP`	`0.29ms/MP`
Simplicidade	Simples de ler e escrever	Muito difícil de ler e escrever. Alto conhecimento de domínio.	**Simples** de ler e escrever. Curva de aprendizado envolvida.
Portabilidade	Facilmente portátil	Não é portátil. Precisa de implementação para hardware diferente.	Oferece **suporte a diferentes hardwares** e pode gerar binários. Podemos definir programações diferentes para hardwares diferentes ou reverter para a computação inline.
Capacidade de manutenção	De fácil manutenção	Difícil de manter, pode ser problemático se os especialistas deixarem a equipe.	**Fácil manutenção**. Necessário conhecimento adicional, mas é mais fácil de aprender do que aprender os intrínsecos de cada hardware.

Fonte: https://blog.minhazav.dev/write-fast-and-maintainable-code-with-halide/

Halide é simples e rápida

Bilateral grid

Reference C++: 122 lines
Quad core x86: 150ms

CUDA C++: 370 lines
GTX 980: 2.7ms

Halide algorithm: 34 lines
schedule: 6 lines
Quad core x86: 14ms

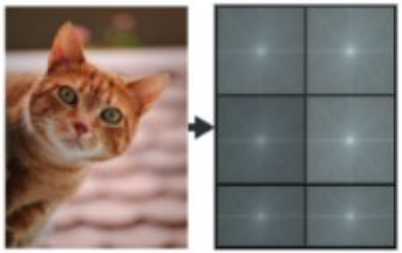
GPU schedule: 6 lines
GTX 980: 2.3ms

Local Laplacian filters

C++, OpenMP+iIPP: 262 lines
Quad core x86: 210ms

Halide algorithm: 62 lines
schedule: 11 lines
Quad core x86: 92ms

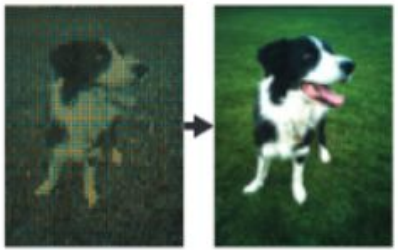
GPU schedule: 9 lines
GTX 980: 23ms



Fast Fourier transform

FFTW: *thousands*
Quad core x86: 384ns
Quad core ARM: 5960ns

Halide algorithm: 350 lines
schedule: 30 lines
Quad core x86: 250ns
Quad core ARM: 1250ns



Camera pipeline

Optimized assembly: 463 lines
ARM core: 39ms

Halide algorithm: 170 lines
schedule: 50 lines
ARM core: 41ms

DSP schedule: 70 lines
Hexagon 680: 15ms

Fonte: https://dl.acm.org/doi/pdf/10.1145/3150211

Leitura recomendada

- ➔ Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing
- ➔ Materiais e vídeos em: https://halide-lang.org/
- ➔ https://www.codee.com/catalog-category/glossary/
- ➔ https://en.wikipedia.org/wiki/Loop_optimization

Obrigado pela atenção!

INSTITUIÇÃO EXECUTORA

Centro de Informática UFPE

FADE UFPE

COORDENADORA

MCTI FUTURO

Softex

APOIO

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÃO

GOVERNO FEDERAL BRASIL UNIÃO E RECONSTRUÇÃO