Die Bildbearbeitungssprache Halide

Dominik Danner

13.06.2013

Inhaltsverzeichnis

Inhaltsverzeichnis

Probleme der Domäne Bildbearbeitung

Ausgangslage:

- zweidimensionale Bilder
- pro Pixel dieselbe Operation (SIMD)

Optimierungsmöglichkeiten:

- parallele Schleifen
- Zwischenspeichern von mehrfach verwendeten Teilergebnissen

Spannweite

Totale Vorberechnung $\Leftrightarrow$ Berechnung in der Schleifeniteration

Beispiel: 3x3 Blur Boxfilter

Der 3x3 Blur Boxfilter berechnet für jeden Bildpunkt den Durchschnittswert seiner Nachbarn.

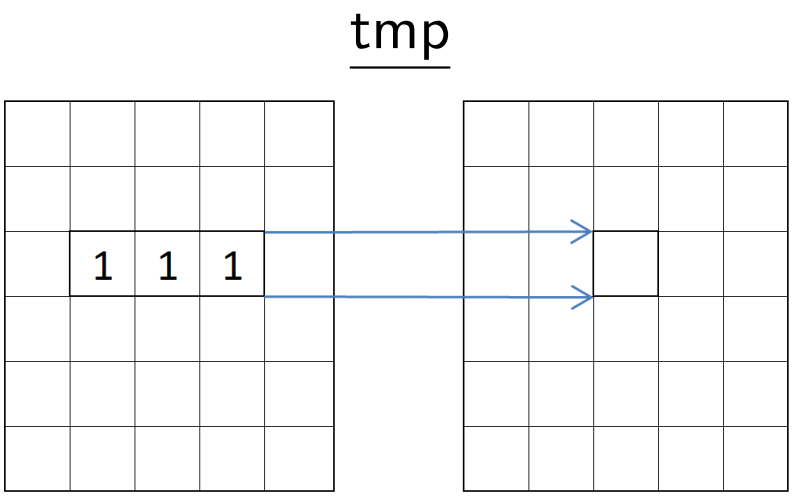
$$blurred(x, y) = \frac{1}{9} \cdot \sum_{i=-1}^{1} \sum_{j=-1}^{1} input(x + j, y + i)$$

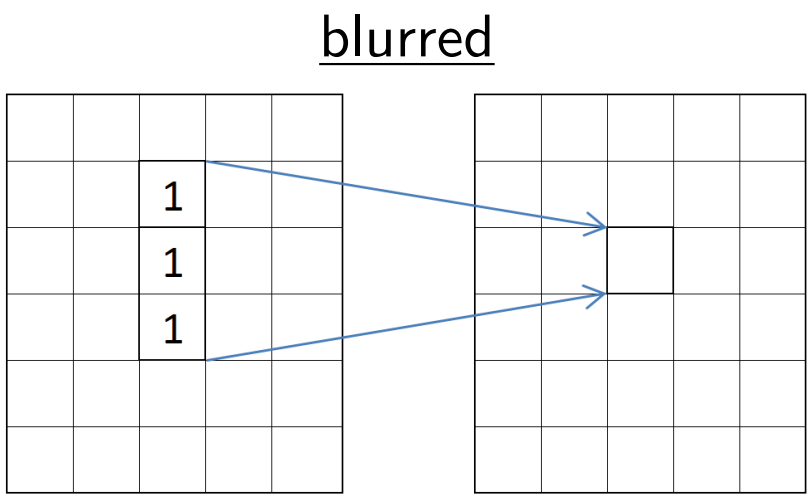
Um die überlappenden Berechnung besser zu sehen:

$$blurred(x, y) = \frac{1}{3} \cdot (tmp(x, y - 1) + tmp(x, y) + tmp(x, y + 1))$$

$$tmp(x, y) = \frac{1}{3} \cdot (input(x - 1, y) + input(x, y) + input(x + 1, y))$$

Veranschaulichung





Implementierung

Schleifenbasiert hat man zwei einfache Möglichkeiten

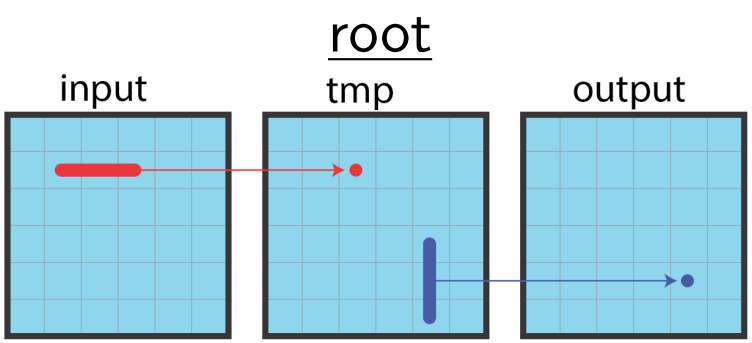
- Totale Vorberechnung von tmp (root):

```
1 for (int y = 0; y < in.height(); y++)
2     for (int x = 0; x < in.width(); x++)
3         tmp(x, y) = (in(x - 1, y) + in(x, y) + in(x + 1, y)) / 3;
4
5 for (int y = 0; y < in.height(); y++)
6     for (int x = 0; x < in.width(); x++)
7         blurred(x, y) = (tmp(x, y - 1) + tmp(x, y) + tmp(x, y + 1)) /
              3;
```

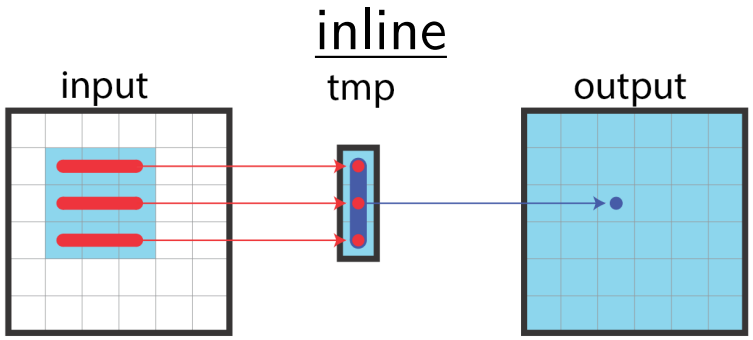
- Berechnung von tmp während der Iteration (inline):

```
1 for (int y = 0; y < in.height(); y++)
2     for (int x = 0; x < in.width(); x++)
3         for(int i = -1; i <= 1; i++)
4             tmp(x, y + i) = (in(x - 1, y + i)
5                            + in(x    , y + i)
6                            + in(x + 1, y + i)) / 3;
7         blurred(x, y) = (tmp(x, y - 1) + tmp(x, y) + tmp(x, y + 1)) /
              3;
```

Implementierung



- keine doppelten Berechnungen
- geringe Producer-Consumer Lokalität



- geringer Zwischenspeicherbedarf
- redundante Berechnungen von tmp

Lösung

Das Optimum liegt irgendwo zwischen root und inline.

Aber

Dies kann bei einfacher C++ Programmierung in langen, unleserlichen Code ausarten, der dann meist Plattform abhängig ist.

Inhaltsverzeichnis

Die Sprache Halide

Allgemeines zu Halide

- Compiler und domänenspezifische Sprache eingebettet in C++
- Anwendungsbereich in der Bildbearbeitung
- Entwickelt am MIT von Jonathan Ragan-Kelley

Compiler für

- x86
- CUDA
- ARM

Ziel von Halide

Mit Halide wollen die Entwickler Effizienz und Portabilität mit geringer Codekomplexität vereinen.

Grundlegende Idee

Grundgedanke

Trennung von Algorithmus und Schedule

- Algorithmus wird funktional geschrieben:
  - Bilder werden als Funktionen betrachtet.
  - Kette von Funktionen bilden den Algorithmus.
- Schedule bestimmt die Durchführung:
  - Berechnungsbeziehung zwischen Funktionen
  - Abarbeitungsreihenfolge der Dimensionen

Algorithmus

Funktionen in Halide bestehen aus:

- logischen oder arithmetischen Ausdrücken,
- bedingten Anweisungen,
- Aufrufen von Funktionen,
- Laden von Bildpunkten,
- Aufrufen von Funktionsargumenten oder festen Werten.

Achtung!

Halide Funktionen haben nur einen Integer Wertebereich.

Halide Code - 3x3 Blur Boxfilter

```
1 tmp(x, y)     = (input(x - 1, y) + input(x, y) + input(x + 1, y)) / 3;
2 blurred(x, y) = (  tmp(x, y - 1) +   tmp(x, y) +   tmp(x, y + 1)) / 3;
```

Schedule - Beziehung von Funktionen

Man kann Funktionen kennzeichnen mit:

- inline (Standard)
- root (.compute_root(), Standard bei der Ausgabefunktion)
- genaue Berechnungsvorschrift (.store_at(), .compute_at())

Schedule - Abarbeitungsreihenfolge

Bei Funktionen, die nicht inline ausgeführt werden, kann das Abarbeitungsschema festgelegt werden.

- sequenzielle Berechnung
- parallele Berechnung
- Ausrollen einer Schleife durch eine Konstante
- Vektorisieren mit konstanter Vektorbreite
- Dimensionen neu ordnen
- Splitten einer Dimension in innere und äußere

Serial y, Serial x

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Serial x, Serial y

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

Serial y, Vectorized x

1	2
3	4
5	6
7	8
9	10
11	12
13	14
15	16

Parallel y, Vectorized x

1	2
1	2
1	2
1	2
1	2
1	2
1	2
1	2

Split x into $2x_o+x_i$,
Split y into $2y_o+y_i$,
Serial y_o, x_o, y_i, x_i

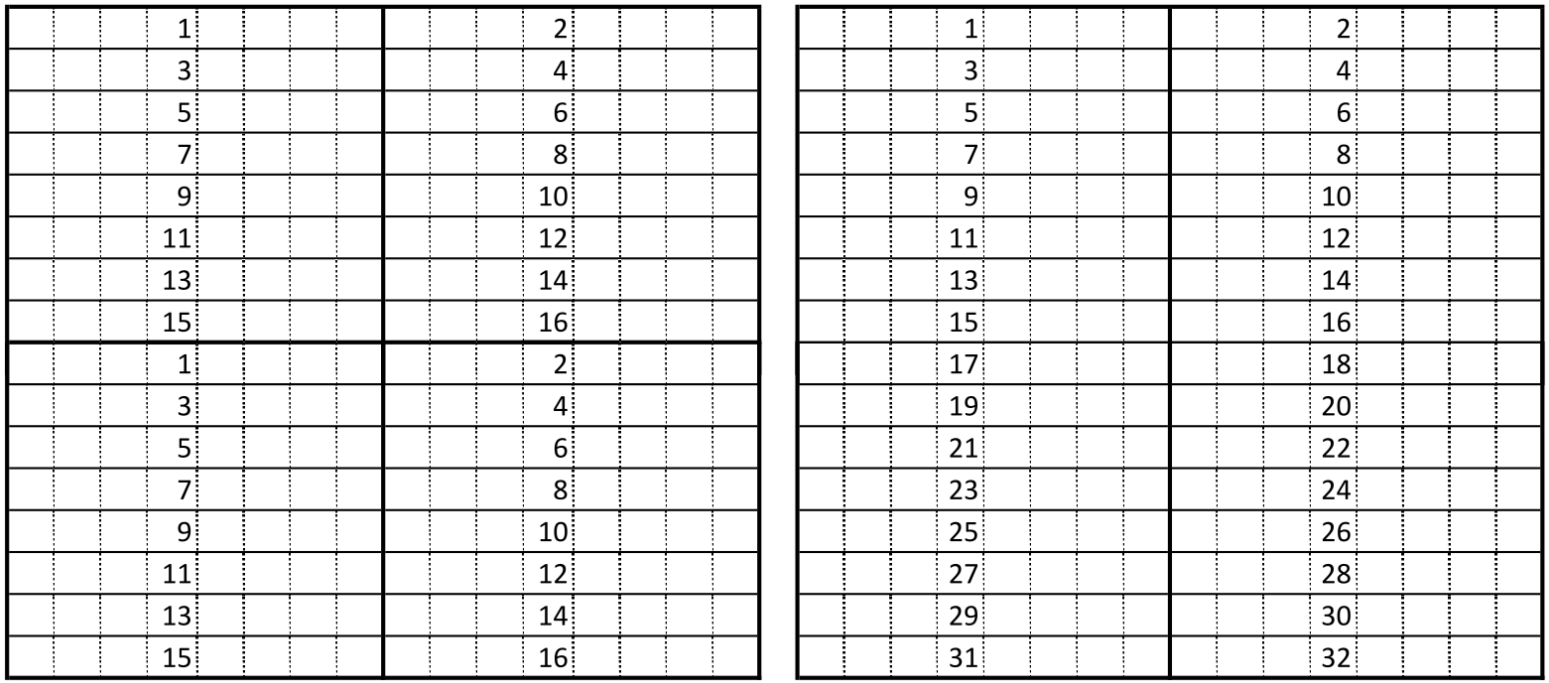
1	2	5	6	9	10	13	14
3	4	7	8	11	12	15	16
17	18	21	22	25	26	29	30
19	20	23	24	27	28	31	32
33	34	37	38	41	42	45	46
35	36	39	40	43	44	47	48
49	50	53	54	57	58	61	62
51	52	55	56	59	60	63	64

Schedule - 3x3 Blur Boxfilter

Ein möglicher Schedule

```
1 blurred.split(y, y, yi, 8).parallel(y).vectorize(x, 8);
2 tmp.store_at(blurred, y).compute_at(blurred, yi).vectorize(x, 8);
```

Visualisierung dieses Schedules



Gesamter Code 3x3 Blur Boxfilter

```
1  #include <Halide.h>
2  using namespace Halide;
3
4  int main(int argc, char **argv) {
5
6      ImageParam input(UInt(16), 2);
7      Func tmp, blurred;
8      Var x, y, yi;
9
10     // The algorithm
11     tmp(x, y) = (input(x - 1, y) + input(x, y) + input(x + 1, y)) / 3;
12     blurred(x, y) = (tmp(x, y - 1) + tmp(x, y) + tmp(x, y + 1)) / 3;
13
14     // How to schedule it
15     blurred.split(y, y, yi, 8).parallel(y).vectorize(x, 8);
16     tmp.store_at(blurred, y).compute_at(blurred, yi).vectorize(x, 8);
17
18     blurred.compile_to_file("halide_blur", input);
19
20     return 0;
21 }
```

Inhaltsverzeichnis

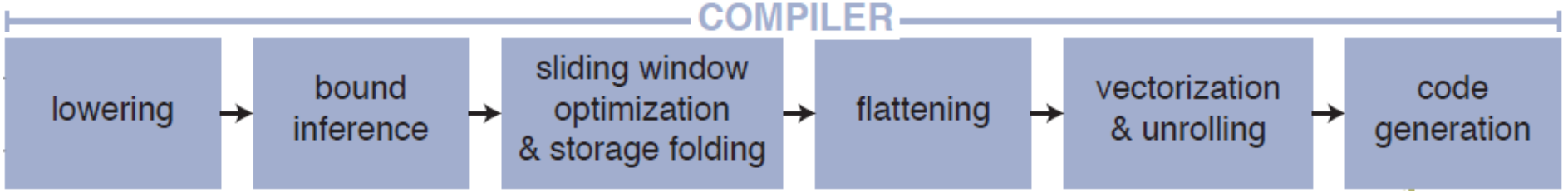
Compiler

Eine Bildbearbeitungsfunktion in Halide kann

- JIT kompiliert und dann ausgeführt werden oder
- statisch kompiliert und durch Objekt und Header Datei für andere Programme bereit gestellt werden.

Allgemeines zum Compiler:

- Algorithmus + Schedule werden zu imperativen Programm
- wandelt eins-zu-eins den Schedule des Programmierers um



Lowering und Bound Inference

Lowering

- Konstruktion einer großen, verschachtelten Schleife um die Funktionen
- Kennzeichnung der einzelnen Schleifen mit vec, for, unroll, par
- Schleifengrenzen vorerst mit abhängigen Platzhaltern

Bound Inference

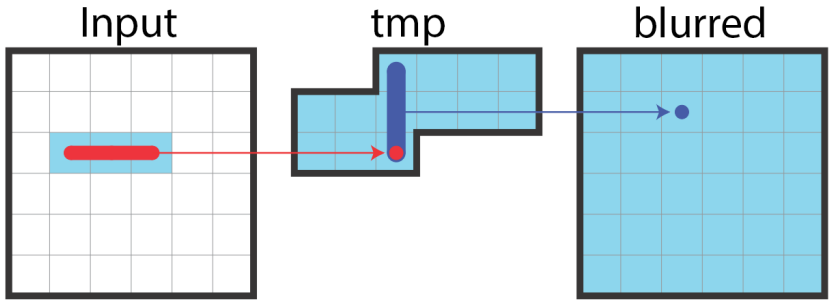
- Schleifengrenzen werden mit Intervallarithmetik ermittelt.
  Bsp: g(x) = f(x + 1) + f(x * 2) auf [2, 4]. Dann f(x) im Bereich [min(2 + 1, 2 * 2), max(4 + 1, 4 * 2)] = [3, 8]
- Einfügen von let- Ausdrücken an den passenden Stellen

Sliding Window, Storage Folding and Flattening

Sliding Window und Storage Folding funktioniert nur dann, wenn

- eine serielle Schleife vorhanden ist und
- Daten außerhalb der Schleife gespeichert und in der seriellen Schleife berechnet werden.

⇒ Berechnungen vorhergehender Iterationen können wiederverwendet werden.



Flattening "plättet" Speicheradressierung auf eindimensionale Ausdrücke.
Bsp: aus f(x, y) wird f(y * width + x)

Vectorization and Unrolling

Schleifen, die mit Vektorisieren oder Unrolling gekennzeichnet sind, haben nach dem Lowering konstante Längen.

Unrolling erzeugt k-mal den Code im Schleifenrumpf mit hochlaufenden Indices.

Vectorization

- ähnlich wie Unrolling
- spezielle Funktion ramp(k)
- In Code Generation wird aus ramp(k) ein Vektorbefehl.

Keine Autovektorisierung!

Vektorisierung muss man immer per Hand im Schedule angeben.

Code Generation

Zuerst noch Check auf:

- “toten” Code
- Constant-folding

Der Backend-Compiler LLVM wandelt den in Schritt 1-5 entstandenen Code in Maschinenbefehle um.
An zwei Stellen verändert:

- Vektorausdrücke
- Parallele Schleifen

Inhaltsverzeichnis

Performance - 3x3 Blur Boxfilter

Vergleich verschiedener Hardware hinsichtlich Performance von Halide

- Laptop 1: Intel Core i5 430M@2x2,26 GHz, 8 GB Ram
- Laptop 2: Intel Core i5 2410M@2x2,3 GHz, 4 GB Ram
- Desktop: AMD Athlon II x4 630@4x2,8 GHz, 6 GB Ram

Entwickler versprechen SpeedUp von 12 im Vergleich zur seriellen Implementierung.

Rechner	seriell	optimiert	Halide
Laptop 2	1.307268	0.138345	0.110435
Laptop 1	1.669196	0.259968 (0.208925)	0.196443
Desktop	1.588815	0.314670	0.255997

Rechner	SpeedUp
Laptop 2	11,8
Laptop 1	8,5
Desktop	6,2

Zeitangaben in Sekunden. Tests ohne Beachtung von Randfällen

Performance - 3x3 Blur Boxfilter

Randfall

Für Randfälle wird in Halide eine clamp() Funktion bereitgestellt.

```
1 clamp(x, 0, max_x);
```

Falls der Wert unter oder über den gegebenen Grenzen (0, max_x) liegt, wird 0 bzw. max_x zurückgegeben.

Tests mit eingebauter clamp() Funktion zeigen aber, dass diese noch nicht sehr effizient implementiert ist.

Overhead

Schedule von Blur genau wie in einer seriellen Schleife $\Rightarrow$ Halide 14 % langsamer

JIT vs statisch

Blur Implementierung auf Farbbilder (hier 4 Megapixel)
Unterschiede:

- deutliche Größenunterschiede 18 MB zu 60 KB
- fast identische Ausführungszeit von ca. 30 ms
- aber: JIT Version benötigt weitere 145 ms für die Kompilation der Halide-Funktion

Ausführung	Zeit
JIT	29,743 ms
statisch	30,623 ms

Grenzen - Funktionen

Achtung!

Halide Funktionen haben nur einen Integer Wertebereich.

```
1 // Ausgabe von g: {0, 0, 0, 1, 1} statt {0, 1/3, 2/3, 1, 4/3}
2 Func f, g;
3 Var x;
4 Expr t = 1/3;
5 f(x) = x;
6 g(x) = f(x * t);
7 Image<float> out = g.realize(4);
```

Einschränkung

Keine Algorithmen auf Floatingpoint möglich

Grenzen - Schedule

Achtung!

Bei Vektorisierung, Unrolling, Split muss auf die passende Breite geachtet werden.

```
1 // Compiler wirft Fehler!!
2 Var x;
3 Func f;
4 f(x) = x * x;
5 f.compute_root().vectorize(x, 4);
6 Image<int> out = f.realize(7);
```

Inhaltsverzeichnis

Fazit

- Installation ist nicht gerade einfach, wegen unzureichender Dokumentation.
- Kann schnell aus den Papers der Entwickler und zahlreichen Beispielprogrammen gelernt werden.
- Ist intuitiv zu programmieren.
- Erleichtert das Durchtesten von verschiedenen Schedules, da nur wenig abgeändert werden muss.

Einsatzbereich: Hauptsächlich Bildbearbeitung