
Dokumentation zur Änderung der Code von Robert Spangenberg

# Befreiung des SGM-Codes von SSE

Abgabetermin:05.2017

**Student:** Min Bao
**Betreuer:** Ruf Boitumelo
76131 Karlsruhe

**Betrieb:**
Fraunhofer IOSB
Fraunhoferstraße 1
76131 Karlsruhe

# Inhaltsverzeichnis

## Abbildungsverzeichnis

## Abkürzungsverzeichnis

| | |
|---|---|
| **SGM** | Semi-Global-Matching |
| **SSE** | Streaming SIMD Extensions |
| **CPU** | Central Processing Unit |

# 1 Einleitung

- Der von Robert Spangenberg geschrieben Code setzt Semi Global Matching auf CPU um. SSE-Befehle werden verwendet, um die Ausführung des Codes zu beschleunigen. Intel führte das 128 Bit breite SSE-Register ein, so dass mit SSE-Befehlen doppelt so viele Daten auf einmal parallel verarbeitet werden können, wie mit dem auf 64-Bit-Registern basierenden Computer. Aber in einem eingebetteten System wird hinsichtlich der Dimension und Spannungsversorgung kein SSE-Register zur Verfügung gestellt, deshalb soll der Code von SSE-Befehlen befreien, damit der in dem eingebetetten System ausführbar ist.

# 2 Census Transformation

- In den Code wird die Census Transformation für ein 5x5 Fenster eingeführt. Der Grauwert des zentralen Pixels $p$ wird einzeln mit seinen Nachbarn $q$ verglichen und das Ergebnis $\sigma(x, y)$ als Binärzahl abgespeichert. Die Funktion ist definiert als:

$$\sigma(x,y) := \begin{cases} 0 & für \quad q \geq p \\ 1 & für \quad q < p \end{cases} \tag{1}$$

  Wenn der Graunwert der Nachbarn größer als oder gleich dem Wert des zentralen Pixels, daraus wird ein Null gespeichert. ansonst Eins. z. B.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 126 | 154 | 67 | 109 | 78 | | 1 | 0 | 1 | 1 | 1 | |
| 208 | 129 | 12 | 89 | 177 | | 0 | 0 | 1 | 1 | 0 | |
| 244 | 189 | 128 | 90 | 225 | $\Longrightarrow$ | 0 | 0 | | 1 | 0 | $\Longrightarrow (10111001100010011000 1101)_b$ |
| 128 | 17 | 54 | 155 | 148 | | 0 | 1 | 1 | 0 | 0 | |
| 210 | 127 | 124 | 168 | 23 | | 0 | 1 | 1 | 0 | 1 | |

Abbildung 1: 5x5 Census Transformation

- Für jedes Pixel wird ein solcher Bitstring berechnet, die Berechnung findet von oben links des Bildes bis unten rechts statt. Das 24 Bit Ergebnis wird anschließend mit dem Datentyp Uint32 abgespeichert, deswegen die erste acht Bit muss auf Null gesetzt werden.

```
1  ...
2    resultByte1 = resultByte1|(pixel7h&B1mask15);
3         uint16 setByte1zero = 255u;
4         resultByte1 = resultByte1& setByte1zero;
5
6         uint16 pixel16v =*(i3 +2);
7    ...
```

Fünf Zeiger $i0, i1, i1, i2, i4$ werden deklariert und zeigen den Anfang der ersten fünf Zeilen des Bildes, dann mithilfe einer for-Schleife werden die fünf Zeiger bis letzten fünf Zeilen zeigen. Der detaillierte Code findet sich im Anhang A.1: Census Transformation auf Seite i.

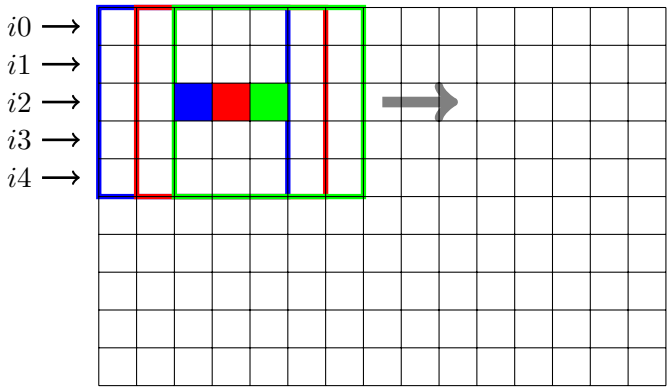


Abbildung 2: Die Berechnung der Census Transformation und das Block schiebt sich nach rechts

Die Census Transformation von den Pixeln in ersten zwei Zeilen als ungültig betrachtet. Der erste gültige Wert befindet sich in der dritten Zeile und dritten Spalte.

```
1  ...
2    uint32* result  = dst + 2*width+2;
3      const uint16* const end_input = src + width*height;
4     ...
```

## 3 Blockmatching und Kosten berechnen

- Um das entsprechende Pixel in rechtem Bild auszufinden, muss das Blockmatching durchgeführt werden. Hiermit wird die Kostenfunktion eingeführt, die ist wie folgendes definiert:

$$C'(x, y, d) := Hamming(P(x, y), Q(x - d + 1, y)) \tag{2}$$

  $P$, $Q$ sind die Ergebnisse von der Census Transformation, in diesem Fall wird Disparität als 128 ($d = 0 \sim 127$) gewählt. Die Kosten von ersten 64 Disparitäten ($d = 0 \sim 63$) werden alle berechnet, alle vierten Kosten der restlichen Disparitäten ($d = 64 \sim 127$) werden abgetastet. Insgesamt werden 80 Kosten für jedes Pixel berechnet. Zwar führt es zur einen kleinen Verschlechterung des Ergebnisses, aber der Code läuft schneller. Das Ergebnis werden in einen Disparity-Würfel gespeichert.

- Bei dem Blockmatching muss man beachten, dass nicht alle Kosten gültig sind. z. B. in Abbildung 3 zeigt. dass nicht alle Pixel im linken Bild entsprechendes Pixel im rechten Bild für alle Dispäritäten haben. Aus der Abbildung 3 geht es aus, dass für dieses Pixel die Kosten in $d = 0 \sim 2$ gültig sind. In $d = 3 \sim 127$ sind die Kosten ungültig, weil die Pixel sich außerhalb des Bildes befinden.

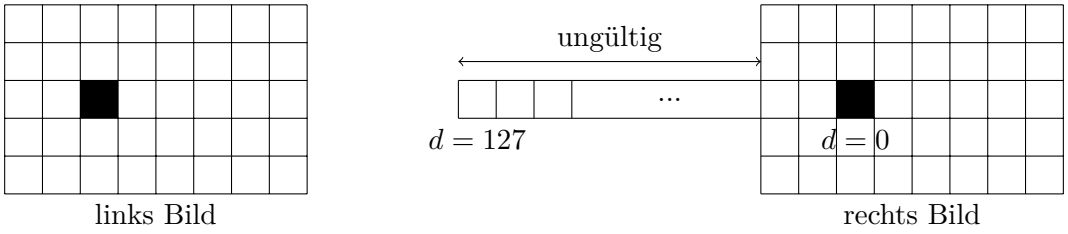


Abbildung 3: Blockmatching

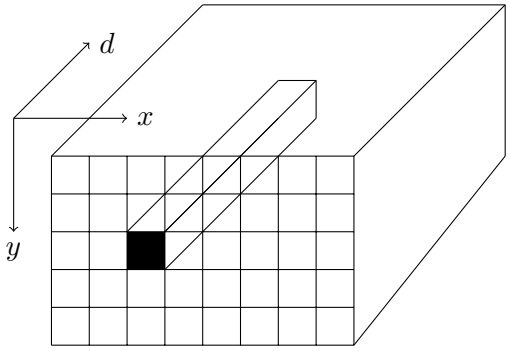


Abbildung 4: Disparity-Würfel

```
1  ...
2    for  (; d >(sint32)j && d >= 0 && d>=dispCountLow; d--) {
3                 sint32 compressedD = dispCountLow+(d-dispCountLow)/dispSubSample;
4                 *getDispAddr_xyd(dsi,width, dispCountCompressed, i,j,compressedD) = invalidDispValue;
5                 pMatchRowJmD++;
6             }
7
8             for  (; d >(sint32)j && d >= 0; d--) {
9                 *getDispAddr_xyd(dsi,width, dispCountCompressed, i,j,d) = invalidDispValue;
10                pMatchRowJmD++;
11            }
12
13    ...
```

Der detaillierte Code findet sich im Anhang A.2: Blockmatching und Kostenberechnung auf Seite iii.

## 4 Kosten aggregieren

- Das Semi Global Matching wird hier vorgestellt, weil pixelweise Kostenberechnung nicht so genau ist und Fehler passieren könnte. Die globale Information im Bild wird gebraucht.Die Technik minimiert die globalen Kosten in horizontaler, vertikaler und in den diagonalen Richtungen.

Wobei 8 oder 16 Pfade verwendet werden können und jeder Kostenpfad $L_r(p,d)$ für ein Pixel $p := (x,y)$ mit Disparität $d$ in Richtung $r$ rekursiv berechnet wird:

$$\begin{aligned} L_r(p,d) =& C(p,d) + \min(L_r(p-r,d), \\ & L_r(p-r,d-1) + P_1, L_r(p-r,d+1) + P_1, \\ & \min_i L_r(p-r,i) + P2) - \min_k L_r(p-r,k) \end{aligned} \tag{3}$$

wobei $P_1$ und $P_2$ Strafen sind, die addiert werden, wenn sich die Disparitäten um 1 bzw. mehr als 1 ändern ($P_1 < P_2$ ). Anhang A.3: Kosten aggregieren auf Seite iv

- Die Aggregation der Kosten fängt von dem ersten Pixel des Würfels an, von oben links bis unten rechts des Bildes, und die läuft durch alle Disparitäten. Bei diesem Scannen aggregieren nur die Kosten der 4 Pfaden: $0, 1, 2, 3$. wie in Abbildung 5 gezeigt wird. Das Ergebnis des ersten Scannen wird zwischengespeichert, dann fängt das zweiten Scannen von unten rechts des Würfels an, das läuft durch alle Dispäritäten bis oben links des Würfels. Die Kosten der restlichen Pfaden $4, 5, 6, 7$ aggregieren, die mit den zwischengespeicherten Werte addieren. Der resultierende Würfel enthält die aggregierten Kosten aller achten Pfaden.

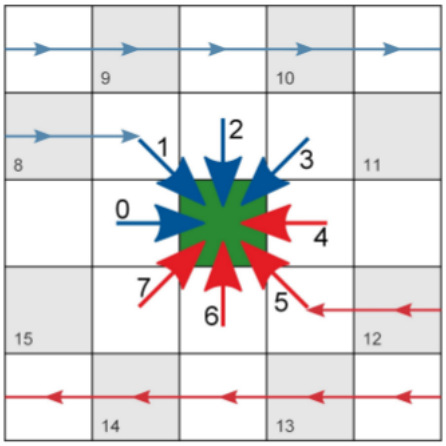


Abbildung 5: Dargestellt sind die 8 Pfade (Ziffern 0-7) beim SGM, deren Pixel für die Kosten des grünen Pixels in der Mitte berücksichtigt werden. Die vier blauen Pfade werden in der ersten (hellblaue Linie) und die roten in der zweiten Phase (hellrote Linie) berechnet. Die grauen Pixel mit den Ziffern 8 bis 15 würden bei 16 Pfaden zusätzlich berechnet werden.

- Um diese Struktur in den Code umzusetzen werden 8 Buffers gebraucht. Die dienen zu dem Speichern aller aggregierten Kosten der 8 Pfaden. Übrigens nicht für alle Pixel des Würfels aggregieren die Kosten der 8 Pfaden. Hier werden ein paar Situationen als Ausnahmen betrachtet. Siehe Abbildung 5.

  1. Die Kosten von dem ersten Pixel sind keine aggregierten Kosten, hier fängt die Aggregation noch nicht an.

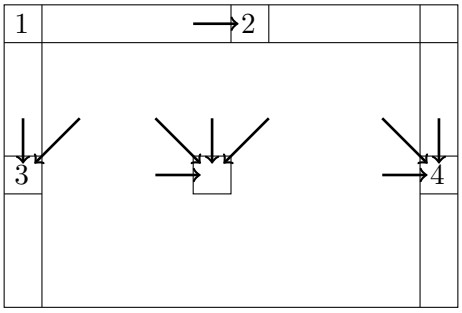


Abbildung 6: Die Ausnahmen beim Aggregation der Kosten (Top-down Scannen)

2. Für die restlichen Pixel in der ersten Zeile aggregieren die Kosten nur von dem Pfad rechts.
3. Für die restlichen Pixel in der ersten Spalte aggregieren die Kosten nur von dem Pfad oben und oben rechts.
4. Die Pixel in der letzten Spalte haben die Kosten, die von dem Pfad rechts, links oben, oben aggregieren.

- Ein paar Dinge sind in den Code zu beachten, 4 Zeiger werden definiert, um die Aggregationskosten von verschiednen Pfaden zu speichern. Das Ergebnis besteht aus der Summe der Aggregationskosten von den 4 Pfaden.(Top-down Scannen).
  Anhang A.4: Summe der Aggregationskosten auf Seite v
  Beim Bottom-up Scannen ist es gleich, dann addieren die 2 Ergebnisse. Lauf der Gleichung 3 geht es aus, wenn die Dispärität $d = 0$ oder $d = d_{\max}$, müssen die Dispäritäten $d = -1$, $d = d_{\max} + 1$ vorher auf einen großen Wert gesetzt werden.

```
1  ...
2    L_r0_last[-1] = L_r1_last[-1] = L_r2_last[-1] = L_r3_last[-1]  = MAX_SGM_COST;
3      L_r0_last[disp] = L_r1_last[disp] = L_r2_last[disp] = L_r3_last[disp] = MAX_SGM_COST;
4      L_r0[-1] = L_r1[-1] =L_r2[-1]=L_r3[-1]= MAX_SGM_COST;
5      L_r0[disp] = L_r1[disp] = L_r2[disp]=L_r3[disp]=MAX_SGM_COST;
6  ...
7    L_r1_last[j*dispP2-1] = L_r2_last[j*dispP2-1] = L_r3_last[j*dispP2-1] = MAX_SGM_COST;
8      L_r1_last[j*dispP2+disp] = L_r2_last[j*dispP2+disp] = L_r3_last[j*dispP2+disp] =
           MAX_SGM_COST;
9
10     L_r1[j*dispP2-1] = MAX_SGM_COST;
11     L_r1[j*dispP2+disp] = MAX_SGM_COST;
12     L_r2[j*dispP2-1] = MAX_SGM_COST;
13   L_r2[j*dispP2+disp] = MAX_SGM_COST;
14     L_r3[j*dispP2-1] = MAX_SGM_COST;
15     L_r3[j*dispP2+disp] = MAX_SGM_COST;
16 ...
```

# A Anhang

## A.1 Census Transformation

```
1
2  for (;  i4+4<=end_input; i0 +=1, i1 +=1 ,i2 +=1, i3 +=1 ,i4 +=1){
3          uint16 pixelcv =*(i2 +2);
4
5          //line1
6          uint16 pixel0h = (*i0  >=pixelcv) ? 0:65535u;
7
8          uint16 pixel1v =*(i0 +1);
9          uint16 pixel1h =(pixel1v >=pixelcv ) ? 0:65535u;
10
11         uint16 pixel2v =*(i0 +2);
12         uint16 pixel2h =(pixel2v >=pixelcv ) ? 0:65535u;
13
14         uint16 pixel3v =*(i0 +3);
15         uint16 pixel3h =(pixel3v >=pixelcv ) ? 0:65535u;
16
17         uint16 pixel4v =*(i0 +4);
18         uint16 pixel4h =(pixel4v >=pixelcv ) ? 0:65535u;
19
20         //line2
21         uint16 pixel5h =(*i1 >=pixelcv ) ? 0:65535u;
22
23         uint16 pixel6v =*(i1 +1);
24         uint16 pixel6h =(pixel6v >=pixelcv ) ? 0:65535u;
25
26         uint16 pixel7v =*(i1 +2);
27         uint16 pixel7h =(pixel7v >=pixelcv ) ? 0:65535u;
28
29         uint16 pixel8v =*(i1 +3);
30         uint16 pixel8h =(pixel8v >=pixelcv ) ? 0:65535u;
31
32         uint16 pixel9v =*(i1 +4);
33         uint16 pixel9h =(pixel9v >=pixelcv ) ? 0:65535u;
34
35         //line3
36         uint16 pixel10h =(*i2  >=pixelcv ) ? 0:65535u;
37
38         uint16 pixel11v =*(i2 +1);
39         uint16 pixel11h =(pixel11v >=pixelcv ) ? 0:65535u;
40
41         uint16 pixel12v =*(i2 +3);
42         uint16 pixel12h =(pixel12v >=pixelcv ) ? 0:65535u;
43
44         uint16 pixel13v =*(i2 +4);
45         uint16 pixel13h =(pixel13v >=pixelcv ) ? 0:65535u;
46
```

```
47          //line4
48          uint16 pixel14h =(*i3 >=pixelcv) ? 0:65535u;
49
50          uint16 pixel15v =*(i3 +1);
51          uint16 pixel15h =(pixel15v >=pixelcv ) ? 0:65535u;
52
53          uint16 B1B2mask0 = 32768u;
54          uint16 B1B2mask1 = 16384u;
55          uint16 B1B2mask2 = 8192u;
56          uint16 B1B2mask3 = 4096u;
57          uint16 B1B2mask4 = 2048u;
58          uint16 B1B2mask5 = 1024u;
59          uint16 B1B2mask6 = 512u;
60          uint16 B1B2mask7 = 256u;
61          uint16 B1mask8 = 128u;
62          uint16 B1mask9 = 64u;
63          uint16 B1mask10 = 32u;
64          uint16 B1mask11 = 16u;
65          uint16 B1mask12 = 8u;
66          uint16 B1mask13 = 4u;
67          uint16 B1mask14 = 2u;
68          uint16 B1mask15 = 1u;
69
70          uint16 resultByte1 = pixel0h & B1mask8;
71          resultByte1 = resultByte1|(pixel1h&B1mask9);
72          resultByte1 = resultByte1|(pixel2h&B1mask10);
73          resultByte1 = resultByte1|(pixel3h&B1mask11);
74          resultByte1 = resultByte1|(pixel4h&B1mask12);
75          resultByte1 = resultByte1|(pixel5h&B1mask13);
76          resultByte1 = resultByte1|(pixel6h&B1mask14);
77          resultByte1 = resultByte1|(pixel7h&B1mask15);
78          uint16 setByte1zero = 255u;
79          resultByte1 = resultByte1& setByte1zero;
80
81          uint16 pixel16v =*(i3 +2);
82          uint16 pixel16h =(pixel16v >=pixelcv ) ? 0:65535u;
83
84          uint16 pixel17v =*(i3 +3);
85          uint16 pixel17h =(pixel17v >=pixelcv ) ? 0:65535u;
86
87          uint16 pixel18v =*(i3 +4);
88          uint16 pixel18h =(pixel18v >=pixelcv ) ? 0:65535u;
89
90          //line5
91          uint16 pixel19h =(*i4  >=pixelcv ) ? 0:65535u;
92
93          uint16 pixel20v =*(i4 +1);
94          uint16 pixel20h =(pixel20v >=pixelcv ) ? 0:65535u;
95
96          uint16 pixel21v =*(i4 +2);
```

```
97          uint16 pixel21h =(pixel21v >=pixelcv ) ? 0:65535u;
98
99          uint16 pixel22v =*(i4 +3);
100         uint16 pixel22h =(pixel22v >=pixelcv ) ? 0:65535u;
101
102         uint16 pixel23v =*(i4 +4);
103         uint16 pixel23h =(pixel23v >=pixelcv ) ? 0:65535u;
104
105         uint16 resultByte2 = pixel8h & B1B2mask0;
106         resultByte2 = resultByte2|(pixel9h&B1B2mask1);
107         resultByte2 = resultByte2|(pixel10h&B1B2mask2);
108         resultByte2 = resultByte2|(pixel11h&B1B2mask3);
109         resultByte2 = resultByte2|(pixel12h&B1B2mask4);
110         resultByte2 = resultByte2|(pixel13h&B1B2mask5);
111         resultByte2 = resultByte2|(pixel14h&B1B2mask6);
112         resultByte2 = resultByte2|(pixel15h&B1B2mask7);
113         resultByte2 = resultByte2|(pixel16h&B1mask8);
114         resultByte2 = resultByte2|(pixel17h&B1mask9);
115         resultByte2 = resultByte2|(pixel18h&B1mask10);
116         resultByte2 = resultByte2|(pixel19h&B1mask11);
117         resultByte2 = resultByte2|(pixel20h&B1mask12);
118         resultByte2 = resultByte2|(pixel21h&B1mask13);
119         resultByte2 = resultByte2|(pixel22h&B1mask14);
120         resultByte2 = resultByte2|(pixel23h&B1mask15);
121
122         uint16* result16 = (uint16*)result;
123         *result16 = resultByte1;
124         *(result16+1) =resultByte2;
125         result =(result+1);
126     }
```

## A.2 Blockmatching und Kostenberechnung

```
1  ...
2      for (sint32 i=lineStart;i < lineEnd;i++) {
3          uint32* pBase = intermediate1+i*width;
4          uint32* pMatchRow = intermediate2+i*width;
5          for (uint32 j=0; j < (uint32)width; j++) {
6              uint32* pBaseJ = pBase + j;
7              uint32* pMatchRowJmD = pMatchRow + j - dispCount +1;
8              sint32 d=dispCount - 1;
9
10             for (; d >(sint32)j && d >= 0 && d>=dispCountLow; d--) {
11                 sint32 compressedD = dispCountLow+(d-dispCountLow)/dispSubSample;
12                 *getDispAddr_xyd(dsi,width, dispCountCompressed, i,j,compressedD) = invalidDispValue;
13                 pMatchRowJmD++;
14             }
15
16             for (; d >(sint32)j && d >= 0; d--) {
```

```
17                  *getDispAddr_xyd(dsi,width, dispCountCompressed, i,j,d) = invalidDispValue;
18                  pMatchRowJmD++;
19              }
20
21              // fill valid disparities
22
23              if (d > dispCountLow - 1) {
24                  for (; d >= dispCountLow;d-= dispSubSample) {
25                      uint16 cost = (uint16)POPCOUNT32(*pBaseJ ^ *pMatchRowJmD);
26                      sint32 compressedD = dispCountLow+(d-dispCountLow)/dispSubSample;
27                      *getDispAddr_xyd(dsi,width, dispCountCompressed, i,j,compressedD) = cost;
28                      pMatchRowJmD += dispSubSample;
29                  }
30                  d+=3;
31                  pMatchRowJmD -= 3;
32              }
33              // align to full-sampled disparities
34
35              for (;d>=0; d--){
36                  uint16 cost = (uint16)POPCOUNT32(*pBaseJ ^ *pMatchRowJmD);
37                  *getDispAddr_xyd(dsi,width, dispCountCompressed, i,j,d) = cost;
38                  pMatchRowJmD++;
39              }
40          }
41      }
```

## A.3 Kosten aggregieren

```
1  ...
2          sint32 minPropCost = L_r0_last[d]; // same disparity cost
3          // P1 costs
4            sint32 costP1m = L_r0_last[d-1]+paramP1;
5            if (minPropCost > costP1m)
6              minPropCost = costP1m;
7                sint32 costP1p = L_r0_last[d+1]+paramP1;
8            if (minPropCost > costP1p) {
9                minPropCost =costP1p;
10               }
11           // P2 costs
12           sint32 minCostP2 = *minL_r0_last;
13           sint32 varP2 = adaptP2(paramAlpha, img_line[j],img_line[j-dj],paramGamma,          paramP2min);
14           if (minPropCost > minCostP2+varP2)
15               minPropCost = minCostP2+varP2;
16         // add offset
17         minPropCost -= minCostP2;
18           const uint16 newCost = saturate_cast<uint16>(cost + minPropCost);
19           L_r0[d] = newCost;
20           if (*minL_r0 > newCost) {
21               *minL_r0 = newCost;
```

```
22            }
23  ...
```

## A.4 Summe der Aggregationskosten

```
1  ...
2    const uint16 newCost = newCost_r0 + newCost_r1 + newCost_r2 + newCost_r3;
3                          //cost sum
4                          if (pass == 0) {
5                              *getDispAddr_xyd(S, width, disp, i,j, d)= newCost;
6                          } else {
7                              *getDispAddr_xyd(S, width, disp, i,j, d) += newCost;
8                          }
9  ...
```