Abschlussbericht - Eingebettete Bildverarbeitung SS 2011
Implementierung eines Streaming-Codecs

Daniel Wäber (4049590) und Alexander Münn (4403061)
Berlin, 7. Oktober 2011

Inhaltsverzeichnis

1. Überblick und Zielstellung

Auf Basis eines XMOS-Mikrocontrollers mit angeschlossenem Kameramodul, haben wir im Rahmen dieses Projekts einen Codec zur Übertragung der Bilddaten entwickelt. Hauptaugenmerk lag dabei darauf, die Komprimierung der Daten auf Streambasis durchzuführen, also den Speicherbedarf aufgrund der Hardwarebeschr̈ankungen weitestgehend zu minimieren.
Abbildung 1.1 zeigt die Formatierung der Bilddaten, wie sie von der Kamera erwartet werden. Die Flags `NEWFRAME` und `NEWLINE` werden mit einer Breite von 32 Bit als `0xFFFFFFFF` oder entsprechend `0xFEFEFEFE` kodiert. Die Implementierung der De- und Encoder sind jedoch von der Kodierung der Kameradaten losgelöst. Um unabhängig von der Kamera-Hardware zu entwickeln, existiert zusätzlich ein in XC geschriebener Testgenerator, der solch ein Datenstream zur Verfügung stellt.

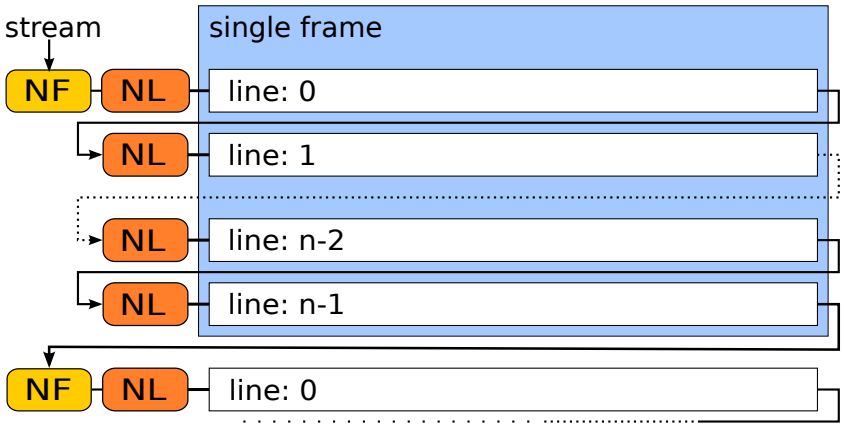

Abbildung 1.1: Formatierung des Kamera-Streams

Um die Integrität des Codecs zwischen Plattformen - z.B. XMOS-Transmitter und UNIX-Receiver - zu gewährleiten, haben wir die De- und Encoder in reinem C geschrieben. Somit konnten wir zusätzlich auf Verwendung Pointern zurückgreifen, was uns einigen redundanten Code ersparrt hat. Infolge dessen mussten wir lediglich APIs für die jeweilige Plattform implementieren.

2. Codec

Die Grundidee der Komprimierung basiert darauf, einen zu kodierenden Pixelwert p_i über ein vorhergehenden (decodierten) Referenzpixel $b_{ref(i,d)}$ anzunähern:

$$b_i = b_{ref(i,dir)} + c_{val} \tag{2.1}$$

Für die eigentlich Kodierung sind an dieser Stelle nur relevant, welchen Wert c_{val} und *dir* annehmen. *dir* steht in diesem Kontext für *direction* und beschreibt mit $ref(i, dir)$ einen Nachbarpixel des Pixel p_i in eine Richtung, die *dir* kodiert. Der Annäherungswert c_{val} *change value* wird implizit bestimmt. Durch die Abweichung

$$d = b_i - p_i \tag{2.2}$$

kann unter der Annahme $b_{i+1} \approx b_i$ eine Entscheidung darüber getroffen werden, ob c_{val} für den nächsten Schritt erhöht oder verringert werden sollte. Diese Information wird im Folgenden als c verstanden. Eine Variante $c_{va]}$ zu aktualisieren zeigt Listing 2.1.

Listing 2.1: Aktualisierung von c_{val} anhand des c-Flags

```
1 if (c) {
      c_val = c_val + c_val/2;
  } else {
      c_val = -c_val;
      if (abs(c_val)>=CMPR_C_MIN*2)
6         c_val /= 2;
  }
```

Daraus lassen sich zwei Eigenschaften ableiten. Zum einen muss c_{val} im laufe der Kodierung mitbestimmt und für jeweilige Referenzpixel gespeichert werden, aber zum anderen verringert sich auch die *distance* d, wenn tatsächlich ähnliche Pixelwerte aufeinander folgen, da sich die rekonstruierten Werte "einschwingen". Faktisch muss bei der Enkodierung eines Pixels p_i folgende Gleichung mit einer Unbekannten (ref) optimiert werden:

$$0 = min|(b + c_{val})_{ref} - p_i| \tag{2.3}$$

Ist ref bestimmt, kann c_{val} konkret angepasst werden. Für die Rekonstruktion sind nur diese beiden Angaben ref und c erforderlich.
In den folgenden Abschnitten sollen zwei implementierte Varianten des Codecs und deren Erweiterung um ein *Run-Length-Encoding* detailierter beschrieben werden.

2.1 Einzelbild-Komprimierung

In der Einzelbild-Komprimierung werden die unmittelbaren Nachbarn des zu kodierenden Pixels als Referenz betrachtet, aufgrund des Streamings natürlich nur der Vorgänger der aktuellen Zeile und der obere Nachbar aus der vorhergehenden Zeile. Dadurch ist es notwendig Daten über letzte Zeile im Speicher zu halten. Abbildung 2.1 zeigt den schematischen Aufbau eines 2D-Encoders, wie er anhand der gespeicherten dekodierten Pixeldaten b_{val} und deren Änderungswerte c_{val} einen Pixel p als *dir* und c enkodiert und intern wieder dekodiert speichert. Im letzten Arbeitsschritt werden b_x und $c_{val,x}$ in den *previous pixel* und an der

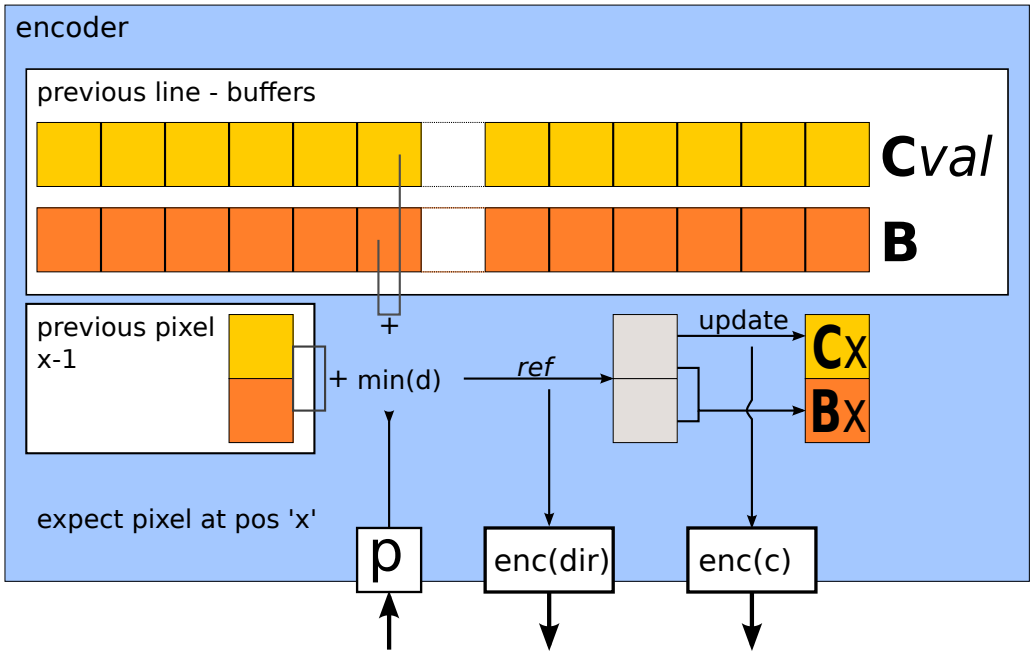

Abbildung 2.1: Schematischer Aufbau eines 2D Encoders

Stelle x in den *line buffer* geschrieben.

Die Ausgabewerte *dir* und c können jeweils in einem Bit kodiert werden. Die Richtung *dir* als `0` für den vertikalen und `1` für den horizontalen Nachbarn. Die Enkodierung c geht bereits aus Listing 2.1 hervor. Ist das Bit gesetzt, wird $c_{va]}$ erhöht, andernfalls verringert. In einem Byte können so vier Pixel kodiert werden. Der Encoder benötigt zusätzlich Anweisungen, wann er seinen Zustand für das nächste Bild bzw. die nächste Zeile anpassen muss. Da am linken und oberen Rand eines Bildes die Vorgängerdaten fehlen, werden hier Standardwerte initialisiert. Im Kompressionsstream werden die Anweisungen `NEWLINE` und `NEWFRAME` dann als Bit-Muster kodiert. Listing 2.2 zeigt die verwendeten Bit-Muster der 2D Kompression und Tabelle 2.1 die daraus resultierende Übertragung.

Listing 2.2: Definierte Bit-Muster in der 2D Kompression

```
#define CMPR_ESCAPE     0xFF
#define CMPR_NEW_LINE   0xFE
#define CMPR_NEW_FRAME  0xFD
```

Bild-Stream	Übertragung
Anweisung: NEWLINE	CMPR_ESCAPE CMPR_NEW_LINE
Anweisung: NEWFRAME	CMPR_ESCAPE CMPT_NEW_FRAME
Pixel	C DIR
4x (C DIR) = CMPR_ESCAPE	CMPR_ESCAPE CMPR_ESCAPE

Tabelle 2.1: Kodierung der Anweisungen NewLine und NewFrame

Ein Bild in der Grösse von 160x120 Pixeln kann so mit etwa $120 \cdot 160/4 + 2 = 5040$bytes kodiert werden. Die Funktionsweise der Kompression lässt bereits erahnen, dass es in Ecken von horizontalen und vertikalen Kanten zu Artefakten kommen wird. In dieser Situation gibt es für den Pixel p_x keine gute Referenz, wodurch sich c_{val} über einige Iterationsschritte anpassen muss. Abbildung 2.2 zeigt solche Artefakte anhand unseres Testvideos. Um diesen Fehler zu minimieren haben wir den Codec im folgenden Abschnitt um die Referenz

Abbildung 2.2: Artefakte in der 2D Komprimierung

zum vorherigen Bild erweitert.

2.2 3D-Komprimierung

Die Coder der 2D-Komprimierung haben wir so erweitert, dass sie zur Rekonstruktion auch Bild-Informationen des vorherigen Bildes speichern. Da b und $c_{va]}$ des gesamten Bildes die Speicherkapazität überschreiten würde, haben wir die Auflösung der Buffer reduziert (Downsampling).
Für die Kodierung *dir* ergibt sich nun die dritte Möglichkeit *last frame*. Prinzipiell werden für diese Information zwei Bit benötigt. In Kombination mit dem *Run-Length-Encoding*, haben wir *dir* mit einem Bit kodiert (siehe 2.3).
Durch Datenverlust kann es passieren, dass der Decoder den Kompressionsstream nicht weiter verarbeiten kann. Aus diesem Grund haben wir hier eine Sync-Anweisung eingeführt, die als `CMPR_FRAME_SYNC 0xFF` kodiert wird. Sie bewirkt das Zurčksetzen der Buffer auf vordefinierte Standardwerten.
In der Implementierung haben wir die Entwicklung der 3D Coder ohne RLE nicht weiter verfolgt. Abbildung 2.3 analog zu Abbildung 2.2 die Kantenartefakte der Kodierung am Testvideo.

Abbildung 2.3: Reduzierung der Artefakte durch 3D Komprimierung

2.3 Run-Length-Encoding

Gedanke hinter dem Run-Length-Encoding ist es, die Richtung des letzten kodierten Pixels zu bevorzugen. Wurde der letzte Pixel beispielsweise mit seinem horizontalen Nachbarn kodiert, erhählt die horizontale Richtung beim Encoding des nächsten Pixels eine höhere Gewichtung. Ziel ist es über den Verlauf einer Zeile so wenig Änderungen des *dir* wie möglich zu erzeugen. In der Konsequenz tritt *dir* derart wiederholt auf, dass es optimal RLE codiert werden kann (Abb. 2.4).

Die Sementik der RLE-Kodierung der Richtung sieht so aus, dass das d-Bit eines Bytes die Änderung der

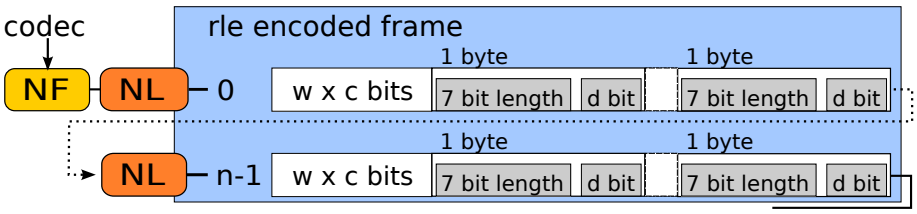

Abbildung 2.4: Kodierung mit Run-Length-Encoding des d

Referenzrichtung nach n Pixeln bewirkt, wobei n in den verbleibenden sieben Bits gespeichert wird. Auf RLE-Basis komprimierte Bilder können so nur noch zeilenweise dekodiert werden. Wie das d-Bit verstanden werden kann, beschreibt die Implementierung in Listing 2.3 in einer Zeile. Die vorherige Richtung muss natürlich bekannt sein.

Listing 2.3: Dekodierung d-Bits eines RLE-Bytes

```
/**
 * Decodes change of directon.
 * parma old old direction* param flag   direction flag
 * return new direction** see cmpr3_encode_dir
 */
static inline int cmpr3_decode_dir(int old, char flag) {
    return flag ? (old == PREVIOUS ? VERTICAL : PREVIOUS)
                : (old == HORIZONTAL ? VERTICAL : HORIZONTAL);
}
```

3. Implementierung

Neben dem Kern des Projektes, dem Kompressions-Algorithmus, haben wir Module zum Ausführen der Kompression und Übertragen der Daten, zum Testen und Analysieren und zur Darstellung des Videostreams am Rechner implementiert und unsere Module mit den Beispielcode für Camera und Netzwerk zusammengesetzt.

Hier wird eine Übersicht über die Aufgaben der Module gegeben und kurz deren Funktionsweise und wichtigen Schnittstellen beschrieben. Weiter Details können aus dem strukturierten und an wichtigen Stellen dokumentierten Quelltext gezogen werden.

Informationen zur Kompilation und Ausführung befinden sich in der dem Projekt beiliegenden `README.txt`.

3.1 Funktionen für Tests und Überprüfungen

Um unsere Ansätze und Implementation analysieren und testen zu können, haben wir Test-Funktionen für Videodaten, die über Channel transportiert werden, geschrieben.

Dies Funktionen befinden sich in `board/test/`.

3.1.1 Generierter Videostream

Die Funktion tst_run_debug_video gibt an einen Channel ein Test-Video aus, indem sich ein Viereck über grauen Hintergrund bewegt. Die Breite und Höhe des generierten Videos kann mit dem vorherigen Aufruf von tst_setup bestimmt werden.

Somit können Methoden schnell und leicht auf dem kleinen, einfachen Teststream analysieren werden.

3.1.2 Videodaten Ausgabe

Für erste Tests gibt die Funktion tst_run_debug_output die Videodaten aus einem Channel über den JTAG-Link in die Standardausgabe aus. Diese einfach und sichere Ausgabe der Videodaten reicht, um Fehler festzustellen. Sie ist allerdings relative langsam und die Darstellung als Zahlen und Buchstaben ungenau und nicht intuitiv erfassbar.

3.1.3 Videodaten Analyse

Die Funktion tst_run_frame_statistics überprüft die Konsistenz des Videos und gibt die Frame-Rate in die Standardausgabe aus.

Syntaxfehler im Videodaten-Format und das Fehlen von Pixel oder Zeilen werden festgestellt und Performce-Tests können mit dieser Funktion durchgeführt werden.

3.1.4 Channel-Daten Ausgabe

Zur Analyse komprimierten Video-Daten schriebt die Funktion tst_run_dump_stream alle Daten eines Streams in Hexadezimal-Darstellung in die Standardausgabe.

3.2 Video-Kompression

Wir haben den Kompressions-/Dekompressions-Algorithmus in C implementiert.

Da dieser nun sowohl auf dem XMOS Board wie auch auf dem empfangenden Rechner verwendet wird, ist somit die Konsistenz zwischen diesen Komponenten sichergestellt.

Zum Anderen konnten wir durch die Verwendung von C-Pointern Code sowohl zwischen De- und Encoder wie auch zwischen 2D- und 3D-Komprimierung wiederverwenden, wodurch wir Wiederholungen im Quelltext vermeiden konnten.

3.2.1 Funktionsweise

Sowohl En- wie Decodierung für 2D und 3D haben den gleichen Aufbau. Die Funktionen cmpr_start_frame und cmpr_start_line setzen die Standard-Werte als Referenz-Werte und setzen Laufvariablen zurück.

Für die Codierung jedes Pixels werden jeweils folgende Schritte durchlaufen:

- alle Referenz-Werte für den aktuellen Pixel aus Speicher des Codecs laden und für Encodierung deren Distanz zum echten Pixelwert berechnen (cmpr_context_load)
- Auswahl der besten Richtung anhand der Distanz bzw. Benutzung der kodierten Richtung, um Werte auszuwählen. (cmpr_context_select_dir)
- den Änderungswert (c) anpassen (cmpr_context_update_c)
- berechnete Werte (b und c) im Speicher des Coders setzen, sodass diese als Referenz für später folgende Pixel benutzt werden können (cmpr_context_store)

Bei der Encodierung werden getroffenen Entscheidungen kodiert und zurückgegeben, bei der Decodierung werden diese Informationen benutzt, um den Pixelwert zu rekonstruieren und zurückzugeben.

Wir verwenden RLE für die Übertragung der Richtung bei der 3D Variante, was zu einen weitern Verarbeitungsschritt nach einer abgeschlossener Zeile führt. Auch wird hier wegen Speicherknappheit für die Referenzwerte des vorherigen Bildes Subsampling verwendet.

3.2.2 Benutzung der Schnittstellen

Ein Benutzung der Kompression befindet sich z.B. in `board/video/compress.c`.

Nach der Initialisierung durch cmpr_init wird der Kodierungsprozess mit cmpr_start_frame und cmpr_start_line vorbereitet.

Bei der 2D Variante kann nun Pixel für Pixel nun mit cmpr_enc bzw. cmpr_dec en/decodiert werden.

Bei der 3D Variante mit RLE müssen zunächst die Pixelwerte für eine Zeile mit cmpr3_enc_push an den Codierer gegeben werden, anschliessend können die kodierten c und d Werte für die Zeile mit cmpr3_enc_get_cs bzw. cmpr3_enc_get_dirs ausgelesen werden. Bei der Decodierung werden diese mit cmpr3_dec_push_cs und cmpr3_dec_push_dir übergeben, woraufhin die Pixelwerte für eine Zeile mit cmpr3_dec_pull ausgelesen werden können.

3.3 Zusammensetzen und Ausführung

Die Aufteilung des Projekts in Module ist in der Abbildung **??** dargestellt. Stream-Transmitter bzw. Receiver führen jeweils die Funktionen zusammen, um den Videostream zu aufzunehmen, kodieren und versenden bzw. empfangen, dekodieren und darzustellen.

```
./
  codec/
  videfs/
  receiver/
  board/
    chksm/
    compat/
    common/
    net/
    cam/
    test/
    stream-transmitter/
  mk/
```

¡+Abbildung: Übersicht der Module+¿

`codec` enthält den in C implementierten Kompressions-Algorithmus, in `videfs` sind Parameter für die Komprimierung und das Video festgelegt. Die Video-Darstellung für das Hostsystem befindet sich in `receiver`. In dem `board` Ordner befindet sich der Quelltext spezifisch für das XMOS Board.

`common`, `chksum` und `compat` enthalten einfach Funktionen, die von anderen Modulen verwendet werden. Das `net`-Module enthält den ethernet Code für den XMOS Chip, `cam` das Kamera-Modul. Unsere Test-Funktionen befinden sich in `test`. Zusammengesetzt werden dies Funktionen in dem `stream-transmitter`-Modul.

3.3.1 Receiver

Der Receiver benutzt zur graphischen Darstellung `gl`, wobei die `glut`-Bibliothek verwendet wird um ein Fenster zu öffnen.

Die über einen UDP-Socket empfangenen Daten werden in der receiver Funktion analysiert und entsprechend des Packet-Headers decodiert. Sobald ein Frame abgeschlossen ist, wird die dargestellte `gl`-Graphik erneuert.

3.3.2 Benutzung von Kamera und Netzwerk

Wir haben den vorhanden Quelltext für Kamera und Netzwerk in dem Projekt verwendet. Allerdings musste dieser noch korrigiert und angepasst werden.

So sendete z.B. das Kameramodul die Video-Daten nicht in dem vereinbartem Format über den Channel. Auch die Frequenz für das Kameramodul muss je nach Rechenbelastung auf dem Chip angepasst werden.

Um den Video-Stream versenden zu können, benutzen wir spezielle Header in den UDP-Packeten. Dazu haben wir den Netzwerkcode in der `udb.xc` Datei entsprechend anpasst.

3.3.3 Kodierung wählen und Parameter anpassen

Die Art der Kodierung kann in der `main.xc` im Ordner `board\streaming-transmitter` angepasst werden. Da jede Kodierung eigene Packet-Header besitzt, kann der Receiver weiterlaufen, während sich die Art der Kodierung ändert.

In der `main.xc` kann auch statt dem Kamerabild das Test-Video verwendet werden.

Andere Parameter werden in der `videfs/config.h` verändert, wonach sowohl Receiver wie auch Transmitter neu compiliert werden müssen.

4. Ergebnisse

4.1 Bildqualität

Hardwarebedingt ist es uns nicht möglich aussagekräftige Bilder gegenüberzustellen. Durch Verringer der Kamerafrequenz wird die Bildqualität bereits erheblich reduziert.

Sowohl bei 2D wie auch bei 3D Komprimierung sind die Artefakte sichtbar, die sich auf dem Bild befindende Objekte sind aber gut erkennbar:

(a) RAW (b) 2D Coder (c) 3D Coder

Abbildung 4.1: Codec-Ausgabe im Vergleich zum RAW-Fromat

4.2 Komprimierung

	bytes/frame
raw	19200
cmpr	5040
cmpr3 (rle)	≈3200

Tabelle 4.1: Gegenüberstellung der Komprimierung

4.3 Performance

	Framerate mit Test-Video		Rate und Frequenz mit Camera
	mit En- und Decodierung	Encodierung und Netzwerk	Encodierung und Netzwerk
raw	133	52	10 (FREQ_DIV = 6)
cmpr	48	54	5 (FREQ_DIV = 12)
cmpr3	26	27	2 (FREQ_DIV = 28)

Tabelle 4.2: Performance der Codecs, alle Angaben in Hz

Leider kann die Performance nicht gut unter echten Bedingungen gemessen werden, da sowohl Cameramodul wie auch das Netzwerk die Datenrate begrenzen. So zeigt nur die En- und Decodierung eines Testvideos auf dem Chip selbst zuverlässig an, wie sich die verschiedenen Kodierungen verhalten.
Sobald die Daten über Netzwerk übertragen werden, stellt dies zumindest bei 2D Komprimierung noch die Begrenzung da.
Der Test auf dem Kamerabild ist nicht aussagekräftig, da das Kameramodul durch die zusätzliche Last nicht schnell genug zuverlässige Bilder liefern kann. Deshalb musste die Kamerafrequenz verändert werden, was hier zur eigentlichen Begrenzung der Framerate führt.