Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Multimediakommunication und Signalverarbeitung

Prof. Dr.-Ing. André Kaup

Forschungspraktikum

C++/MATLAB-Schnittstellen für die HEVC-Referenzsoftware

Michael Filler

März 2015

Betreuer: Dipl.-Ing. Andrea Eichenseer

Inhaltsverzeichnis

Kapitel 1

Einführung

In vielen Forschungsgebieten gibt es eine Implementierung eines Standards, wie z.B der High Efficiency Video Coding(HEVC) Standard[1], der in C/C++ geschrieben worden ist. Natürlich kommt es vor, dass man in der Forschung diese Implementierung gerne anpassen würde an sein eigenes Forschungsgebiet. Eine Möglichkeit ist es, den C/C++ Code anzupassen. Leider ist es in manchen Projekten nicht so einfach, da der Code sehr komplex ist und kaum bzw. keine Code-Dokumentation mit sich bringt. Auch ist das Debuggen nicht so schön und einfach wie man es z.B in MATLAB gewohnt ist. Deswegen wird in diesem Forschungspraktikum ein Weg gesucht, mit dem es möglich ist mit MATLAB implementierten Code in das C/C++ Projekt einzufügen. Die Funktion die der MATLAB Code erledigen soll ist es, die Referenzbilder eines Frames zu manipulieren. Die Referenzbilder sind die vorhergehenden Bilder mit denen das aktuelle Bild verglichen wird um Bewegungen im Bild festzustellen um effizienter übertragen zu können. Dies ist in diesem Fall besonders interessant, da es sich um Bilder einer Fischaugenkamera handelt die runde Bilder macht und somit das translatorisches Bewegungsmodell nicht gilt. Deshalb funktionieren die Bewegungserkennungs-Algorithmen des HEVC nicht gut und es ist notwendig das die Referenzbilder anzupassen, um die Algorithmen zu unterstützen. In dieser Arbeit werden der MATLAB Coder™ und die MATLAB

[1] Version 16.2[1]

Engine vorgestellt die beide diese Aufgabe erfüllen.

Kapitel 2

Verschiedene Schnittstellen

2.1 MATLAB Coder™[3]

2.1.1 Erklärung

Der MATLAB Coder™erzeugt C/C++-Code, den man anschließend in sein Projekt einbinden kann. Ein anderer Nutzen des Coders ist, dass er auch MEX-Funktionen generieren kann, die das Ausführen von rechenaufwändigen Abschnitten des MATLAB-Codes beschleunigen können. In dieser Arbeit ist der Geschwindigkeitsvorteil des C/C++-Codes oder der MEX Funktionen nicht interessant, sondern viel mehr wie einfach man damit arbeiten kann. Zunächst wird jedoch die Benutzung des Coders beschrieben.

2.1.2 Benutzung

In MATLAB

Um den Coder zu öffnen, wählt man in MATLAB APPS in der oberen Menüleiste aus und klickt schließlich auf MATLAB Coder. Rechts erscheint ein neues Fenster, welches in Fig. 2.1 abgebildet ist. Zunächst muss man eine *.m*-Datei auswählen, die als Einstiegspunkt für den erzeugten Code dient. In diesen Fall wurde bereits die Da-

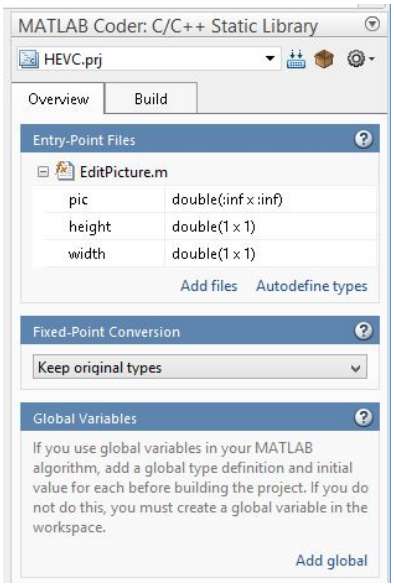

Abbildung 2.1: MATLAB Coder Fenster

tei *EditPicture.m* ausgewählt. Die Übergabeparameter, welche hier *pic*, *height* und *width* heißen, muss man mit den in MATLAB üblichen Typen und Größen angeben. Unter dem Build-Reiter findet man noch weitere Einstellungen, wie die Ausgabedatei und den Ausgabetyp. Für den Ausgabetyp kann man z.B. eine MEX-Funktion, C/C++ Static Library oder C/C++ Dynamic Library auswählen. In der hier beschriebenen Implementierung wurde als Typ C/C++ Static Library ausgewählt. Weiterhin muss eingestellt werden, dass C++ Code ausgegeben wird. Ein wichtiger Punkt ist die Benennung des Ausgabeprojektes. Da das Ziel eine statische Library wird, sollte der Ausgabename *lib* als Prefix haben.

Durch den *Build* Vorgang werden verschiedene *.cpp*, *.h* und andere Dateien erstellt, die notwendig sind um den Code auszuführen. Um den erstellten Code in einem Projekt auszuführen sind weitere Schritte notwendig, welche im Folgenden speziell in der oben genannten Referenzimplementierung des HEVC gezeigt werden.

Im HEVC

Makefile Eine der erzeugten Dateien heißt NameDesProjektes_rtw.mk. Diese Datei ist die Makefile, welche mit

```
make -f NameDesProjektes_rtw.mk
```

erstellt werden kann. Diese Makefile erstellt dann die Datei NameDesProjektes.a, welche in das Verzeichnis $HEVC/lib$ kopiert werden muss. Diesen Vorgang kann man automatisieren indem man in die Hauptmakefile, welche im Verzeichnis $HEVC/build/linux$ zu finden ist, folgendes hinzufügt:

```
$(MAKE) -C ../../Pfad/zur/Lib -f NameDesProjektes_rtw.mk
cp ../../Pfad/zur/Lib/NameDesProjektes.a ../../lib
```

Der nächste Schritt ist die Library an den Compiler zu linken. Dazu muss man die drei Makefiles der Programme, welche man unter HEVC/build/linux/app/$Programm$/ findet, abändern. In jeder der drei Makefiles müssen die Variablen

- DYN_DEBUG_LIBS
- $STAT\_DEBUG\_LIBS$
- $DYN\_RELEASE\_LIBS$
- $STAT\_RELEASE\_LIBS$

angepasst werden.

Jeder Variable muss am Ende ein $-lNameDesProjektes$ angehangt werden. Dabei ist zu beachten, dass das Prefix *lib* weggelassen werden muss. Eine Zeile schaut dann also wie folgt aus:

```
DYN_DEBUG_LIBS = -lTLibEncoder$(HBD)d
    -lTLibCommon$(HBD)d
    -lTLibVideoIO$(HBD)d
    -lTAppCommon$(HBD)d
    -lNameDesProjektes
```

Besonders wichtig ist hierbei, dass die neue Library als letztes hinzugefügt wird, da es sonst zu Problemen kommen kann.

Code Um den erstellten Code auszuführen muss man an entsprechender Stelle eine Header - Datei einbinden:

```
#include "../Pfad/zur/Lib/NameDesProjektes.h"
```

Die Funktion (hier EditPicture) die aufgerufen wird, hat wie in Fig. 2.1 gezeigt drei Eingabeparameter und einen Ausgangsparameter. Die Eingangs- und Ausgangsmatrix wird als emxarray_real_T* gespeichert und zusammen mit zwei Int's der Funktion übergeben. Die Initialisierung der Objekte und Funktionsaufruf schaut dann wie folgt aus:

```
//Codes Ausschnitt aus TComSlice.cpp::
// DoSomeMatlabStuff(TComPic* pcPic)
Int iWidthAll = 0;
Int iHeightAll = 0;
for(Int chan=0; chan<pcPic->getNumberValidComponents();
 chan++) {
  const ComponentID ch=ComponentID(chan);
  const Int iWidth = pcPic->getPicYuvRec()->getWidth(ch);
  const Int iHeight = pcPic->getPicYuvRec()->getHeight(ch);
  if(iWidthAll == 0) {
    iWidthAll = iWidth;
 }
iHeightAll += iHeight;
}
emxArray_real_T *MATLABInput;
emxArray_real_T *MATLABOutput;
MATLABInput = emxCreate_real_T(iHeightAll,iWidthAll);
MATLABOutput = emxCreate_real_T(iHeightAll,iWidthAll);
```

```
19 MakeDataForMATLAB(pcPic, MATLABInput);
20 EditPicture(MATLABInput, iHeightAll,
21     iWidthAll, MATLABOutput);
22 SetDataFromMATLAB(pcPic, MATLABOutput);
```

In der Funktion MakeDataForMATLAB A.1 werden die Daten von einem Bild in die Variable MATLABInput geschrieben und in der Funktion SetDataFromMATLAB A.2 werden die Daten von MATLAB wieder zurückgeschrieben.

2.2 MATLAB Engine[2]

2.2.1 Erklärung

Die MATLAB Engine Bibliothek beinhaltet Funktionen, die es ermöglichen, MATLAB aus einem z.B in C/C++ geschriebenem Programm aufzurufen. Dazu wird eine vollständig installierte Version von MATLAB benötigt. Das besondere hierbei ist, dass man alle möglichen MATLAB Befehle im C/C++ Code ausführen kann, im Gegensatz zum MATLAB Coder, wo man nur einen begrenzen Befehlssatz hat. Natürlich ist die Ausführungszeit des erzeugten Codes des Coders jedoch geringer. Da die MATLAB Engine im Hintergrund arbeitet ist es leider nicht möglich eine aufgerufene Funktion in MATLAB zu debuggen. Mit verschiedenen Plots oder der *save* Funktion kann man sich aber gut aushelfen. Wie die MATLAB Engine in ein Unix System integriert wird werde ich im Folgenden aufzeigen.

2.2.2 Benutzung

In MATLAB

In MATLAB muss man nichts unternehmen. Wenn man eine selbstgeschriebene Funktion aufrufen möchte ist es nur wichtig sie im richtigen Verzeichnis abzulegen.

Allgemein

Bevor der Code der MATLAB Engine ausgeführt werden kann müssen ein paar Schritte unternommen werden.
Das wichtigste ist, dass man ein MATLAB Engine Programm nur kompilieren und starten kann, wenn man sich in der C-Shell befindet. Diese kann durch den Aufruf

```
csh
```

im Terminal gestartet werden. Zusätzlich müssen noch zwei Variablen gesetzt werden:

```
set path = ( $path <matlabRoot>/bin )
setenv LD_LIBRARY_PATH <matlabRoot>
 /bin/glnxa64:<matlabRoot>/sys/os/glnxa64
```

<matlabRoot> entspricht dem Verzeichnis in dem MATLAB installiert wurde. An den Pfad gelangt man durch die Eingabe von *pwd* in MATLAB (im CIP: /mnt/SOFT/-FORALL/matlabR2013b). Die Benutzung der MATLAB Engine wird nun in einem kleinem Test - Projekt vorgestellt.

Test - Projekt

Der Code der Test Datei „test.cpp“ schaut wie folgt aus:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <iostream>
#include "engine.h"
#define  BUFSIZE 256

int main()

{
 Engine *ep;
```

```
12  mxArray *T = NULL, *result = NULL;
13  char buffer[BUFSIZE+1];
14  double time[10] = { 0.0, 1.0, 2.0, 3.0, 4.0, 5.0,
15   6.0, 7.0, 8.0, 9.0 };
16
17  if (!(ep = engOpen(""))) {
18   fprintf(stderr,
19    "\nCan't start
20 MATLAB engine\n");
21   return EXIT_FAILURE;
22  }
23  T = mxCreateDoubleMatrix(1, 10, mxREAL);
24  memcpy((void *)mxGetPr(T), (void *)time,
25    sizeof(time));
26  engPutVariable(ep, "T", T);
27  engEvalString(ep, "D = .5.*(-9.8).*T.^2;");
28  engEvalString(ep, "plot(T,D);");
29  engEvalString(ep, "title('Position vs.
30 Time for a falling object');");
31  engEvalString(ep, "xlabel('Time (seconds)');");
32  engEvalString(ep, "ylabel('Position (meters)');");
33
34  printf("Hit return to continue\n\n");
35  fgetc(stdin);
36
37  return EXIT_SUCCESS;
38 }
```

Mit dem Befehl *engOpen* wird das MATLAB Engine Objekt geöffnet, mit welchem die Funktionen arbeiten werden. Die Variable T wird mit der Funktion $mxCreateDoubleMatrix$ erstellt, welcher die Dimensionen der Matrix übergeben werden und ob es sich um rea-

le oder komplexe Zahlen handelt. Die Funktion *mxGetPr* liefert den Pointer zurück welcher auf die Daten weißt. In diesen Pointer kann man die verschiedenen Daten eintragen, oder man kopiert sie direkt mit *memcpy* hinein. *engPutVariable* fügt eine als Parameter übergebene Variable in den MATLAB Workspce ein, welche dann mit verschiedenen Befehlen, die mit *engEvalString* ausgeführt werden, verändert werden kann. Mit *engEvalString* ist es möglich jede MATLAB Funktion aufzurufen, wie hier z.B *plot*.

Um diese Datei zu kompilieren kann man folgende Makefile benutzen:

```
1 all:
2  g++ -o test test.cpp -I/mnt/SOFT/FORALL/
3   matlabR2013b/extern/include/
4  -L/mnt/SOFT/FORALL/matlabR2013b/bin/
5   glnxa64 -leng -lm -lmat -lmx -lut
```

Führt man das erstellte Programm aus, erhält man genau dieselben Ausgaben, wie man es aus MATLAB heraus erhalten würde.

HEVC

Makefile Genauso wie bei dem MATLAB Coder müssen auch hier verschiedene Makefiles des HEVC angepasst werden. Die drei Makefiles der verschiedenen Programme, welcher unter HEVC/build/linux/app/*Programm*/ zu finden sind müssen wieder angepasst werden. Zunächst fügt man eine Variable der Makefile hinzu, welche das Verzeichnis und die Links zu den Bibliotheken beinhaltet.

```
1 MATLAB_SPECIAL = -L/mnt/SOFT/FORALL/matlabR2013b/bin/
2  glnxa64 -leng -lm -lmat -lmx -lut
```

Auch die Variablen

- *DYN_DEBUG_LIBS*
- *STAT_DEBUG_LIBS*

- $DYN\_RELEASE\_LIBS$
- $STAT\_RELEASE\_LIBS$

müssen wieder angepasst werden.

Jeder Variable muss am Ende ein $(MATLAB_SPECIAL) angehängt werden. Eine Zeile schaut dann also wie folgt aus:

```
DYN_DEBUG_LIBS = -lTLibEncoder$(HBD)d
    -lTLibCommon$(HBD)d
    -lTLibVideoIO$(HBD)d
    -lTAppCommon$(HBD)d
    $(MATLAB_SPECIAL)
```

Da alle Veränderungen im Code in der TComSlice Klasse stattfinden, muss in der Makefile welche die Klasse kompiliert, auch noch Änderungen vorgenommen werden. Der Variable $USER\_INC\_DIRS$ muss der Ort an dem sich die Header Dateien befinden angehängt werden:

```
USER_INC_DIRS = -I$(SRC_DIR) -I/mnt/SOFT/FORALL/
 matlabR2013b/extern/include
```

Da diese Variable bereits verwendet wird, sind sonst keine weiteren Änderungen notwendig.

Sind alle beschriebenen Änderungen ausgeführt, kann man das Projekt kompilieren.

Code Die einzige Änderung die im Code gemacht werden muss ist es, die *engine.h* header Datei in die gewünschte Datei (hier TComSlice.h/cpp) einzufügen.

```
#include "engine.h"
```

Wenn der Header eingefügt ist, kann man wie im obigen Beispiel einfach ein Engine-Objekt aufmachen und mit diesem Arbeiten.

Um das Referenzbild zu laden und wieder zu speichern kann der Code wie folgt aussehen:

```
Int iWidthAll = 0;
Int iHeightAll = 0;
for(Int chan=0; chan<pcPic ->getNumberValidComponents();
  chan++) {
  const ComponentID ch=ComponentID(chan);
  const Int iWidth  = pcPic->getPicYuvRec()->getWidth(ch);
  const Int iHeight = pcPic->getPicYuvRec()->getHeight(ch);
  if(iWidthAll == 0) {
    iWidthAll = iWidth;
 }
iHeightAll += iHeight;
}
Engine *ep;
mxArray *T = NULL;
if (!(ep = engOpen(""))) {
   fprintf(stderr, "\nCan't start MATLAB engine\n");
    throw;
}

T = mxCreateDoubleMatrix(iWidthAll, iHeightAll, mxREAL);
double  *Data = mxGetPr(T);
MakeDataForMATLABEngine(pcPic, Data);
engPutVariable(ep, "T", T);
engEvalString(ep, "T = T'");
engEvalString(ep, "imshow(T,  [0 255])");
engEvalString(ep, "save('img', 'T');");
T = engGetVariable(ep, "T");
Data = mxGetPr(T);
SetDataFromMATLABEngine(pcPic, Data);
```

Der Code schreibt die Pixelwerte in ein Array mit der Funktion MakeDataForMATLABEngine (siehe A.3), setzt den Pointer T in den MATLAB Workspace, transponiert das Bild, da das Pixel zeilenweise und nicht spaltenweiße ausgelesen werden. Das transformierte Bild wird geplottet und in einer .mat Datei gespeichert. Zuletzt wird die geänderte Variable wieder in die Variable des C++ Codes mit der Funktion SetDataFromMATLABEngine (siehe A.4) gespeichert.
Sofern man das Programm über die Shell startet funktioniert alles. Führt man das Programm jedoch in Eclipse aus, muss noch eine Einstellung angepasst werden.

Eclipse Die Umgebungsvariablen PATH und LD_LIBRARY_PATH müssen in Eclipse explizit gesetzt werden, da die Umgebungsvariablen der Shell nicht übernommen werden. Dies kann unter den Projekteinstellungen (Project->Properties->C/C++ Build->Variables) vornehmen. Die tatsächlichen Werte auf die man die Variablen setzen muss, können mit

```
1 echo $PATH
2 echo $LD_LIBRARY_PATH
```

herausgefunden werden.

2.3 Fazit

Mit beiden Varianten kann man MATLAB Code in C/C++ ausführen. Mit dem MATLAB Coder hat man den Vorteil, dass man direkten C/C++ Code bekommt, der sehr eine niedrigere Ausführungszeit als MATLAB hat. Der Nachteil ist jedoch, dass man eine eingeschränkte Funktionalität hat. Mit der MATLAB Engine kann man jede mögliche MATLAB Funktion aufrufen, jedoch wird dadurch das Programm sehr langsam und träge. Je nachdem welche Anforderungen gestellt sind können beide Varianten Sinn machen.

Kapitel 3

Zusatzaufgabe: Optimierung von MATLAB Code

Eine weitere Aufgabe im Praktikum war es MATLAB-Code zu optimieren. In diesem Code ging es um Motion Estimation und Compensation von Fischaugensequezen. Da es zu weit führen würde, den ganzen Code hier aufzulisten stelle ich im Folgenden alle Stellen vor, in dem ich den Code optimiert habe:

- Vektorisierung von Schleifen (auch noch an zwei weiteren Stellen)

```
[numy, numx, ~] = size(xcur_cut);
curVal = zeros(numy, numx);
for itera = 1: numx*numy
    curVal(itera) = img(ycur_cut(itera), ...
     xcur_cut(itera));
end
```

⇓

```
[numyq, numxq, ~] = size(xcur_cut);
curVal = img(ycur_cut(1:numxq*numyq) + ...
```

```
(xcur_cut(1:numxq*numyq)-1) * ...
size(img,1));
```

- Auslagern von langen Abschnitten in eigene Funktionen

```
ssd = inf;
mvssd = [NaN NaN];
for mvx = -searchrange:sr_step:searchrange
 for mvy = -searchrange:sr_step:searchrange
  xccur_cut_undist_m = ...
   xccur_cut_undist + mvx;
  yccur_cut_undist_m = ...
   yccur_cut_undist + mvy;
  ...
  ...
 end
end
```

$$\Downarrow$$

```
[ssd,mvssd,compSSDVal] = Search(searchrange,   ...
sr_step, xccur_cut_undist, ...
yccur_cut_undist, ...
flp, upfactor, refup, imcenter,curVal);
```

Durch diese Optimierungen kann die Laufzeit um etwa $\frac{1}{6}$ gesenkt werden.

Literaturverzeichnis

[1] Fraunhofer HHI. High Efficiency Video Coding (HEVC) Repository 16.2. https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/tags/HM-16.2/.

[2] MathWorks. Introducing MATLAB Engine. http://www.mathworks.com/help/matlab/matlab_external/introducing-matlab-engine.html.

[3] MathWorks. MATLAB Coder. http://www.mathworks.com/products/matlab-coder/?s_iid=ovp_prodindex_3584908777001-81870_pm.

Anhang A

Code

A.1 MakeDataForMATLABCoder

```
1  Void    TComSlice::MakeDataForMATLABCoder(TComPic* pcPic,
2   emxArray_real_T *Input) {
3   UInt Index = 0;
4   for(Int chan=0; chan<pcPic ->getNumberValidComponents();
5    chan++)
6   {
7   const ComponentID ch=ComponentID(chan);
8   const Pel* pOrg =  pcPic->getPicYuvRec()->getAddr(ch);
9   const Int iStride = pcPic->getStride(ch);
10  const Int iWidth  = pcPic->getPicYuvRec()->getWidth(ch);
11   const Int iHeight = pcPic->getPicYuvRec()->getHeight(ch);
12
13  for(Int y = 0; y < iHeight; y++ ) {
14      for(Int x = 0; x < iWidth; x++ ) {
15       Intermediate_Int iTemp = (Intermediate_Int) pOrg[x];
16      Input->data[Index++] = iTemp;
17    }
```

```
18      pOrg += iStride;
19    }
20  }
21 }
```

A.2 SetDataFromMATLABCoder

```
1  Void TComSlice::SetDataFromMATLABCoder(TComPic* pcPic,
2  emxArray_real_T *Input) {
3  UInt Index = 0;
4  for(Int chan=0; chan<pcPic->getNumberValidComponents();
5   chan++) {
6  const ComponentID ch=ComponentID(chan);
7  Pel*  pOrg =  pcPic->getPicYuvRec()->getAddr(ch);
8  const Int iStride = pcPic->getStride(ch);
9  const Int iWidth = pcPic->getPicYuvRec()->getWidth(ch);
10 const Int iHeight = pcPic->getPicYuvRec()->getHeight(ch);
11 for(Int y = 0; y < iHeight; y++ ){
12 for(Int x = 0; x < iWidth; x++ ){
13   pOrg[x] = Input->data[Index++];
14  }
15 pOrg += iStride;
16  }
17 }
18 }
```

A.3 MakeDataForMATLABEngine

```
1 Void TComSlice::MakeDataForMATLABEngine(TComPic* pcPic,
2 double *Input) {
3 UInt Index = 0;
```

```
 for(Int chan=0; chan<pcPic ->getNumberValidComponents();
  chan++) {
 const ComponentID ch=ComponentID(chan);
  const Pel* pOrg = pcPic->getPicYuvRec()->getAddr(ch);
  const Int iStride = pcPic->getStride(ch);
  const Int iWidth = pcPic->getPicYuvRec()->getWidth(ch);
  const Int iHeight = pcPic->getPicYuvRec()->getHeight(ch);
      for(Int y = 0; y < iHeight; y++ ) {
        for(Int x = 0; x < iWidth; x++ ) {
          Intermediate_Int iTemp =
           (Intermediate_Int) pOrg[x];
          Input[Index++] = iTemp;
        }
        pOrg += iStride;
      }
  }
}
```

A.4 SetDataFromMATLABEngine

```
Void TComSlice::SetDataFromMATLABEngine(TComPic* pcPic,
      double *Input) {
  UInt Index = 0;
  for(Int chan=0; chan<pcPic ->getNumberValidComponents();
     chan++){
   const ComponentID ch=ComponentID(chan);
   Pel* pOrg = pcPic->getPicYuvRec()->getAddr(ch);
   const Int iStride = pcPic->getStride(ch);
   const Int iWidth = pcPic->getPicYuvRec()->getWidth(ch);
   const Int iHeight = pcPic->getPicYuvRec()->getHeight(ch);

```

```
12    for(Int y = 0; y < iHeight; y++ ){
13      for(Int x = 0; x < iWidth; x++ ){
14        pOrg[x] = Input[Index++];
15      }
16      pOrg += iStride;
17    }
18  }
19 }
```