

# **camera\_rotate**

## **(Laborprojekt)**

*Vorlesung: Labor Rechnersysteme*

**Erik Alexander Hennig**

tae@erik-hennig.me

Technische Akademie Esslingen



# Inhaltsverzeichnis

1 Aufgabenstellung .....	1
2 Implementierung .....	1
2.1 Parameter .....	2
2.1.1 Pragmas .....	2
2.1.2 INPUT_PTR_WIDTH, OUTPUT_PTR_WIDTH und TYPE .....	2
2.1.3 COLS und ROWS .....	2
2.1.4 NPC und TILE_SIZE .....	3
2.2 Probleme .....	4
2.2.1 Dokumentation .....	4
2.2.2 Fehlermeldungen .....	5
2.2.3 Rotation um 0 Grad .....	5
3 Performance .....	5
4 Fazit .....	6
5 Bibliographie .....	7

## 1 Aufgabenstellung

Ziel des Laborprojekts ist die eigenständige Implementierung einer Bildverarbeitung in Hardware. Als Rahmenbedingung gegeben ist die Verwendung des Kria KV260 Vision AI Starter Kits sowie die Nutzung der Xilinx Toolchain für High-Level-Synthese. Darüber hinaus ist die Wahl des Projekts offen. In diesem Fall wurde als selbstgestellte Aufgabe gewählt, ein aufgenommenes Kamera-Bild unter Zuhilfenahme eines Orientierungssensors so zu rotieren, dass es immer aufrecht angezeigt wird. Dabei soll die Rotation performant genug sein, um ein flüssiges Live-Bild anzusehen.

## 2 Implementierung

Um das Performance-Ziel zu erreichen, soll die Rotation des aufgenommenen Bilds in Hardware implementiert werden. Die übrige Logik wird nur in Software implementiert. Abbildung 1 zeigt die architekturelle Umsetzung des Projekts. Der Mikrocontroller-Baustein in der Mitte stellt den nicht-programmierbaren Teil des Kria-Boards dar. Dieser kommuniziert mit dem FPGA-Teil via AXI und AXI-Lite. Die Aufnahme des (verdrehten) Live-Bilds erfolgt mit einer per USB angeschlossenen Webcam. Zur Bestimmung der Kameraorientierung wird ein modernes Smartphone genutzt. Solche Geräte verfügen in der Regel alle über einen Orientierungssensor und sind leicht zu integrieren, da sie über zahlreiche Wege kommunizieren können. Das Smartphone wird mittels einiger Gummibänder mechanisch an die Webcam gekoppelt, sodass beide die gleiche räumliche Orientierung aufweisen. Zum Auslesen der aktuellen Orientierung implementiert das Programm auf dem Kria-Board in einem extra Thread einen minimalistischen Webserver. Der in der Website enthaltene Javascript-Code liest den Orientierungssensor des Smartphones aus und sendet ihn per HTTP zurück zum Webserver. Anhand der erhaltenen Orientierung kann dann das Kamerabild auf dem FPGA um die passende Gradzahl rotiert werden. Die Ausgabe des Bildes erfolgt per HDMI auf einem externen Display oder per SSH mittels X11 Display Forwarding auf einem weiteren PC.

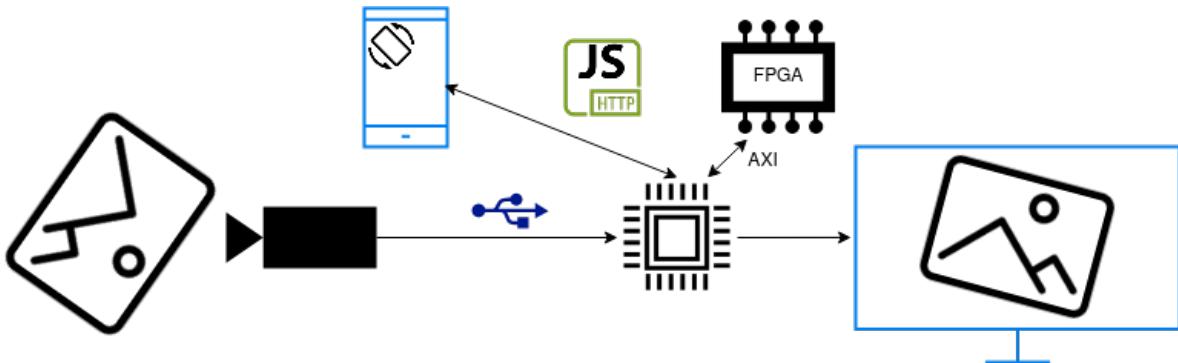


Abbildung 1: Kommunikationskanäle

Zur Umsetzung der Rotation in Hardware kommen folgende Funktionen der Vitis Vision Library auf den ersten Blick in Betracht:

- `remap` [1]: vertauscht Pixel anhand einer Relokationsmatrix
- `warpTransform` [2]: wendet eine affine Transformation auf das Eingangsbild an
- `rotate` [3]: rotiert das Eingangsbild um 90, 180 oder 270 Grad

Die ersten beiden Funktionen erlauben eine beliebige Rotationen. Allerdings sind sie nicht ganz so trivial zu verwenden wie `rotate`, das explizit nur für die Rotation implementiert wurde. Aus diesem Grund wurde `rotate` für die erste Implementierung genutzt und dann aus Zeitgründen auch nicht mehr ersetzt.

## 2.1 Parameter

In diesem Abschnitt werden auf die Template-Parameter der genutzten Funktion `xf::cv::rotate` der Vitis Vision Library eingegangen sowie die genutzten Pragmas für die Code-Generierung der High-Level-Synthese beschrieben.

### 2.1.1 Pragmas

Die Argumente für `rotate` wurden größtenteils einfach durchgereicht. Dabei wurde die Übergabe kleiner Argumente per AXI-Lite umgesetzt. Dies betrifft die tatsächliche Breite und Höhe des Bilds, den Rotationswinkel und die Steuerung des IP-Blocks. Die Übertagung der Eingabe- und Ausgabe-Bilddaten erfolgt über einen AXI-Bus<sup>1</sup>. Dabei wurden folgende Parameter gewählt:

- `offset=slave`: Die Vitis-Dokumentation schreibt diesen Wert vor [4]
- `depth=__XF_DEPTH`: Repräsentiert die Größe des Adressbereichs [4], also die maximale Größe des Bilds in Bytes (= Höhe \* Breite \* Kanäle \* Kanalbreite =  $512 * 512 * 1 * 1 = 262144$ )
- `bundle=gmem0/bundle=gmem1`: Eingabe- und Ausgabebild werden über getrennte Bundles übertragen, um Performance-Einbüßen zu vermeiden [5].

Die Auflistung der INTERFACE-Pragmas sieht somit folgendermaßen aus:

```
#pragma HLS INTERFACE mode=s_axilite port=rows
#pragma HLS INTERFACE mode=s_axilite port=cols
#pragma HLS INTERFACE mode=s_axilite port=direction
#pragma HLS INTERFACE mode=s_axilite port=return

#pragma HLS INTERFACE mode=m_axi depth=__XF_DEPTH bundle=gmem0 port=src_ptr
offset=slave
#pragma HLS INTERFACE mode=m_axi depth=__XF_DEPTH bundle=gmem1 port=dst_ptr
offset=slave
```

### 2.1.2 INPUT\_PTR\_WIDTH, OUTPUT\_PTR\_WIDTH und TYPE

Der Template-Paramter `TYPE` gibt die Anzahl an (Farb-)Kanälen und die Bitbreite eines Kanals an. `INPUT_PTR_WIDTH` und `OUTPUT_PTR_WIDTH` geben ebenfalls die Breite eines einzelnen Eingabe- bzw. Ausgabepixels in Bit an. Unglücklicherweise klärt die Dokumentation die Doppelung im Pixelformat und daraus die resultierende Einschränkungen nicht auf. Mit den Werten `INPUT_PTR_WIDTH=OUTPUT_PTR_WIDTH=8` und `TYPE=XF_8UC1` wird die Funktion korrekt ausgeführt, aber andere Kombinationen führen bereits in der C-Simulation zu Segmentierungsverletzungen und Assertion-Fehlern, z.B. `INPUT_PTR_WIDTH=8, OUTPUT_PTR_WIDTH=16, TYPE=XF_8UC1`. Die Nutzung von 3-Kanal-Farbbildern mit `TYPE=XF_8UC3` scheint ebenfalls nicht möglich. Bei einer Eingabe-/Ausgabebreite von 8 Bit ergibt sich ein Assertionfehler und 24 Bit sind bereits laut Dokumentation untersagt, da nur 2er-Potenzen erlaubt sind.

### 2.1.3 COLS und ROWS

Diese Template-Parameter geben die Maximalbreite und -höhe des zu drehenden Bildes in Pixel an. Diese sind in einem gewissen Rahmen frei-wählbar, allerdings muss bei nicht-quadratischen Bildern darauf geachtet werden, die Maße des Ausgabebildes abhängig von der Rotation anzupassen. Andernfalls werden die Pixel durch fehlerhafte Interpretation der Pixelposition an der falschen Stelle dargestellt wie in Abbildung 2 zu sehen.

---

<sup>1</sup>Wie in Abschnitt 2.1.3 beschrieben, erfolgt der Bildzugriff per Memory Mapping. Somit scheinen nur die Zeiger übertragen zu werden. Aus Zeitgründen war eine tiefere Analyse nicht mehr möglich.



Abbildung 2: Rotation eines nicht-quadratischen Bildes

Wird `COLS` und `ROWS` zu groß gewählt, kommt es zu einer Segmentierungsverletzung in der Co-Simulation, obwohl die C-Simulation weiterhin erfolgreich durchgeführt werden kann. Die Gründe hierfür sind leider unklar, aber mittels binärer Suche (`hls/bin_search.py`) konnte für quadratische Bilder die Maximalhöhe/-breite von 515 Pixeln ermittelt werden.

Überraschenderweise ist der Ressourcenverbrauch von `rotate` unabhängig von der Maximalgröße des Bildes. Tabelle 1 zeigt dies beispielhaft für zwei Größen. Dies zeigt, dass es keinen Buffer für das gesamte Bild gibt, obwohl eine Rotation die Pixel nicht lokal begrenzt bewegt, sondern über die komplette Fläche verschiebt, z.B. findet sich das Pixel oben links nach einer Rotation um 180 Grad unten rechts wieder. Der Grund dafür liegt laut Dokumentation darin, dass `rotate` und einige andere Funktionen mittels Memory-Mapping implementiert sind [6], sodass sie direkt auf den RAM des festverdrahteten Prozessors zugreifen müssten.

Seitenlänge	BRAM	DSP	FF	LUT	URAM
512x512px	4(=1%)	10(=0%)	7303(=3%)	12513(=10%)	0
300x200px	4(=1%)	10(=0%)	7303(=3%)	12513(=10%)	0

Tabelle 1: Ressourcenverbrauch bei Variation der maximalen Bildgröße (`NPC=1, TILE_SIZE=32`)

#### 2.1.4 NPC und TILE\_SIZE

Die Parameter `NPC` und `TILE_SIZE` kontrollieren den Ablauf des Algorithmus. `NPC` bestimmt dabei die Anzahl an Pixel, die pro Zyklus abgearbeitet werden. Erlaubte Werte nach Dokumentation sind eins oder zwei, allerdings schlägt eine Assertion in der C-Simulation fehl, wenn der Wert zwei genutzt wird: `ERROR: Hi(15)out of bound(8) in range()`. Aus Zeitgründen konnte dieses Phänomen nicht näher untersucht werden. Das Problem tritt jedoch auf, obwohl die in der Dokumentation aufgelistete Vorbedingung, dass `ROW` und `COL` Vielfache von `NPC` sein müssen, erfüllt ist. `TILE_SIZE` beschreibt, wie viele Pixel als Gruppe verarbeitet werden sollen. Dieser Parameter kann problemlos variiert werden, um einen passenden Trade-Off zwischen Ressourcen-Verbrauch und Performance zu finden. Abbildung 3 zeigt, dass zwar die minimale Latenz unabhängig von der `TILE_SIZE` ist, aber ihr Durchschnitt und Maximum sehr wohl davon beeinflusst werden. Neben den abgebildeten Messungen wurde auch die Latenz für eine `TILE_SIZE` von 2 ermittelt. Diese ist jedoch um ein Vielfaches schlechter als der Rest, weshalb sie hier nicht dargestellt ist, um die Skala lesbar zu halten. Das Initiation-Interval unterscheidet sich nur geringfügig von der Latenz und wird deshalb ebenfalls nicht dargestellt.

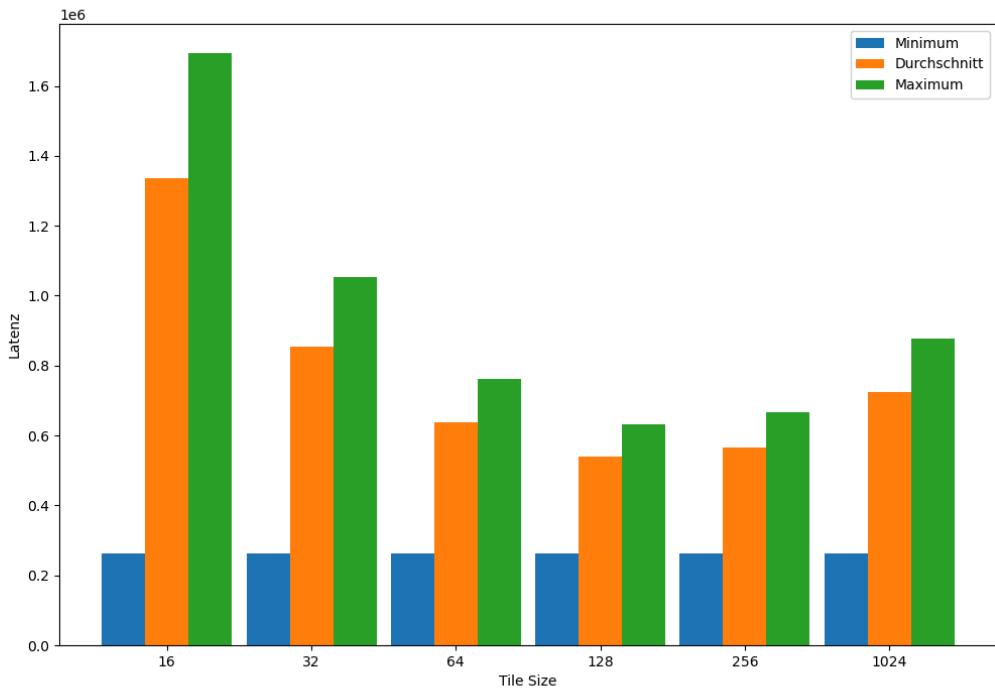


Abbildung 3: Latenz in Abhängigkeit vom Parameter TILE\_SIZE

Das performance-technische Optimum scheint sich bei einer `TILE_SIZE` von circa 128 zu finden. Abbildung 4 stellt den Ressourcenverbrauch dagegen. Es zeigt sich, dass dieser weitgehendst unabhängig von der `TILE_SIZE` ist. Einzig die benötigten Block-RAMs korrelieren mit ihr. Eine `TILE_SIZE` von 1024 ist nicht möglich, hier mehr als das siebenfache der verfügbaren RAMs benötigt würden. Allerdings werden bei 128 nur 11% der Block-RAMs genutzt. Insofern scheint dieser Wert tatsächlich eine Art Optimum für diese Implementierung darzustellen.

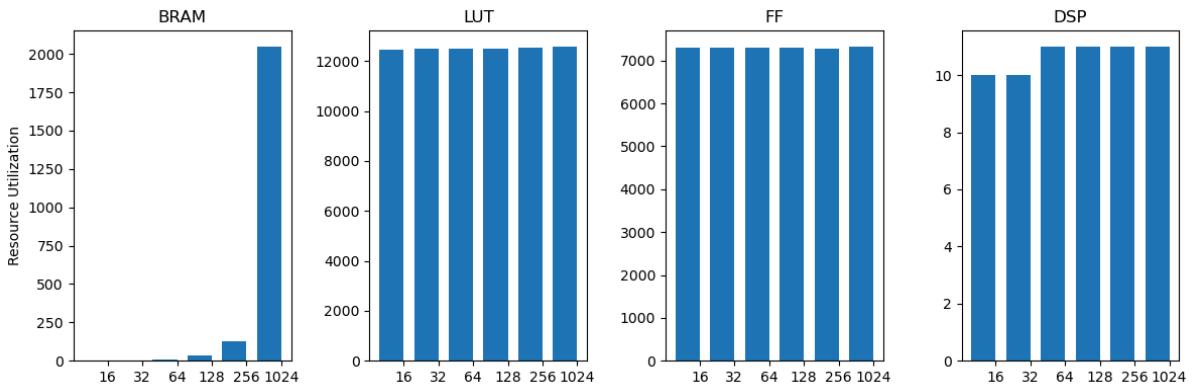


Abbildung 4: Ressourcenverbrauch in Abhängigkeit vom Parameter TILE\_SIZE

## 2.2 Probleme

Der folgende Abschnitt beschreibt kurz aufgetretene Probleme bei der Implementierung der High-Level-Synthese-Funktionalität des Projekts.

### 2.2.1 Dokumentation

Die Dokumentation der Vitis Vision Library hat an einigen Stellen zu Problemen geführt. Beispielsweise beschreibt sie, welcher Header für welche Funktion zu inkludieren ist, separat und

für manche Funktionen, wie z.B. die genutzte `xf::cv::rotate`, fehlt ein Eintrag [7]. Somit war nicht direkt ersichtlich, was der korrekte Header ist. Ein weiteres Problem war die fehlerhafte Dokumentation von `rotate`. Als erlaubte Werte für den `direction` Parameter werden explizit 90, 180 und 270 genannt [3]. Bei Aufruf mit diesen Parameter ergibt sich jedoch immer eine Rotation um 90 Grad. Nach einiger Suche im Quellcode der Vision Library findet sich die schuldige Codestelle:

```
// xf_rotate.hpp:69
for (int k = 0; k < NPC; k++) {
    #pragma HLS UNROLL
    if(direction == 0){
        // rotation by 270 degrees
    }
    else if(direction == 1){
        // rotation by 180 degrees
    }
    else {
        // rotation by 90 degrees
    }
}
```

Somit sind die korrekten Parameter-Werte nicht 90, 180 und 270, sondern 2, 1 und 0, was der Dokumentation widerspricht. Mit diesen neuen Werte funktioniert die Rotation in eine beliebige der drei möglichen Richtungen.

### 2.2.2 Fehlermeldungen

Eine weitere Herausforderung sind die Fehlermeldungen. Wenn beispielsweise die `INPUT_PTR_WIDTH` keine 2er-Potenz ergibt, so wie in der Dokumentation vorgeschrieben, dann werden bei der C-Simulation viele Warnungen ausgegeben, dass Streams keine Daten enthalten, und es kommt schließlich zum Laufzeitfehler. Gerade als Anfänger in der Welt der High-Level-Synthese sind die Warnungen nicht sonderlich verständlich und es ist beinahe unmöglich aufgrund dieser Fehler im Template-Parameter zu finden. Ein weiteres Beispiel für die nicht hilfreiche Fehlermeldung ist auch der Segmentierungsfehler, wenn die Maximalgröße zu groß gewählt wird. Die Fehlermeldung empfiehlt, den Fehler in der Co-Simulation nochmals in der C-Simulation zu beobachten, allerdings tritt er dort nicht auf.

### 2.2.3 Rotation um 0 Grad

Die `rotate` Funktion unterstützt lediglich Rotationen um 90, 180 und 270 Grad. Es ist nicht möglich, keine Rotation durchzuführen, wie aus dem Codeausschnitt in Abschnitt 2.2.1 ersichtlich wird. Um die Eingabe-Bilddaten trotzdem in den Puffer für die Ausgabe-Bilddaten zu übertragen, werden im Fall, dass keine Rotation erwünscht ist, die Daten einfach per `for`-Schleife kopiert:

```
if (rotation == None) {
    for (uint32_t i = 0; i < rows * cols; ++i) {
        dst_ptr[i] = src_ptr[i];
    }
} else {
    xf::cv::rotate/* ... */(src_ptr, dst_ptr, rows, cols, rotation);
}
```

## 3 Performance

TODO

## 4 Fazit

Zusammenfassend lässt sich sagen, dass die praktische Umsetzung eines High-Level-Synthesen-Projekts sehr lehrreich war. Schwierig gestaltet sich vor allem, dass die Dokumentation teilweise unvollständig oder fehlerhaft ist sowie dass die Fehlermeldungen von Xilinx nichts immer hilfreich sind. Von der Performance-Seite her hat sich gezeigt, dass . **TODO**

- performance? measure first? premature opt = root of evil

Die vollständige Implementierung des Projekts findet sich unter [https://github.com/ede1998/camera\\_rotate](https://github.com/ede1998/camera_rotate) [8].

## 5 Bibliographie

- [1] AMD Xilinx, „Vitis Vision Library API Reference: Remap“. [Online]. Verfügbar unter: [https://xilinx.github.io/Vitis\\_Libraries/vision/2022.1/api-reference.html#remap](https://xilinx.github.io/Vitis_Libraries/vision/2022.1/api-reference.html#remap)
- [2] AMD Xilinx, „Vitis Vision Library API Reference: Remap“. [Online]. Verfügbar unter: [https://xilinx.github.io/Vitis\\_Libraries/vision/2022.1/api-reference.html#warp-transform](https://xilinx.github.io/Vitis_Libraries/vision/2022.1/api-reference.html#warp-transform)
- [3] AMD Xilinx, „Vitis Vision Library API Reference: Remap“. [Online]. Verfügbar unter: [https://xilinx.github.io/Vitis\\_Libraries/vision/2022.1/api-reference.html#rotate](https://xilinx.github.io/Vitis_Libraries/vision/2022.1/api-reference.html#rotate)
- [4] AMD Xilinx, „Vitis High-Level Synthesis User Guide: Offset and Modes of Operation“. [Online]. Verfügbar unter: <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Offset-and-Modes-of-Operation>
- [5] AMD Xilinx, „Vitis High-Level Synthesis User Guide: Offset and Modes of Operation“. [Online]. Verfügbar unter: [https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/M\\_AXI-Bundles](https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/M_AXI-Bundles)
- [6] AMD Xilinx, „xf::cv::Mat Image Container Class: Class Definition“. [Online]. Verfügbar unter: [https://xilinx.github.io/Vitis\\_Libraries/vision/2022.1/api-reference.html#id99](https://xilinx.github.io/Vitis_Libraries/vision/2022.1/api-reference.html#id99)
- [7] AMD Xilinx, „Using the Vitis vision Library“. [Online]. Verfügbar unter: [https://xilinx.github.io/Vitis\\_Libraries/vision/2022.1/using-the-vitis-vision-library.html#id1](https://xilinx.github.io/Vitis_Libraries/vision/2022.1/using-the-vitis-vision-library.html#id1)
- [8] E. Hennig, „camera\_rotate“. [Online]. Verfügbar unter: [https://github.com/ede1998/camera\\_rotate](https://github.com/ede1998/camera_rotate)