BERGISCHE
UNIVERSITÄT
WUPPERTAL

Fakultät für Elektrotechnik,
Informationstechnik und Medientechnik

Lehrstuhl für Hochfrequenzsysteme in der Kommunikationstechnik

Team Praktikum

Eingebettete Bildinterpolation von THz Video-Kameras (C-Programmierung).

Team

- EL OUAHABI SAIF EDDINE, 2227087
- BEQQALI AYYOUB, 2227078

Wuppertal, den 09.09.2024

Betreuer: Herr Prof. Dr. rer. nat. Pfeiffer, Ullrich

Inhaltsverzeichnis:

Abbildungsverzrichnis:

1. Einleitung:

Die heutigen Terahertz-Technologien versprechen ein rasches Wachstum in den neuen Märkten wie Medizin- und Sicherheitstechnik, zerstörungsfreie Prüfung, Qualitätskontrolle und Gesundheitsvorsorge. Jedoch ist die Geschwindigkeit, mit der neue Märkte entstehen noch langsam, weil verfügbare Technologien hohen Kosten und eine niedrige Ebene der Integration aufweisen, die ihren Masseneinsatz wesentlich einschränken.

Im Lehrstuhl Lehrstuhl für Hochfrequenzsysteme in der Kommunikationstechnik beschäftigten wir uns intensiv mit der eingebetteten Bildverarbeitung von Terahertz (THz) Video-Kameras. Konkret untersuchten wir die bereits entwickelte und hergestellte THz-Kamera T-LITE M5STICK Plus der Firma M5Stack Technology Co., Ltd., die mit einem THz-Sensor des Typs MLX90640 ausgestattet ist.

Ziel unserer Arbeit war es, die Bilddaten der Kamera zu interpolieren, die von einer Auflösung von 32x24 Pixeln auf eine höhere Auflösung von 135x240 Pixeln skaliert werden sollten. Dies ermöglichte den Vergleich verschiedener Interpolationsmethoden auf zwei unterschiedlichen Plattformen, dem M5Core2 (240x320 Pixel) und dem M5Stick Plus.

Darüber hinaus bestand eine zentrale Aufgabe darin, den Interpolationsalgorithmus aus dem Hauptquellcode des Produkts zu isolieren und separat auf beiden Geräten zu implementieren. Abschließend untersuchten wir die Bildwiederholrate (FPS) für verschiedene Algorithmen auf beiden Plattformen, um die Effizienz und Leistungsfähigkeit der Implementierungen zu vergleichen.

2. Hardware:

Im rahmen diesem Praktikum haben wir uns mit Zwei Hardware Beschäftigt: M5STACK CORE2, und T-LITE M5StickC-PLUS.

2.1 M5Stack Core2:

Der M5Core2 ist die zweite Generation der M5Stack-Entwicklungskits, die die Funktionen der ersten Generation von Kernen weiter verbessert.

Abbildung 2-1: M5Stack Core2 [1].

Der Core2 für AWS ist mit einem ESP32-D0WDQ6-V3-Mikrocontroller ausgestattet, der zwei Xtensa 32-Bit-LX6-Kerne und eine Hauptfrequenz von bis zu 240 MHz sowie 2,4-GHz-Wi-Fi bietet. Er hat 8MB PSRAM und 16MB Flash an Bord.

2.2 T-Lite M5StickC-PLUS:

T-Lite ist ein integriertes Wärmebildaufnahme-Temperaturmessgerät mit Online-Erkennungsfunktion, das Produkt besteht hauptsächlich aus M5StickC-PLUS und UNIT THERMAL, mit MLX90640-BAA Infrarot-Bildsensor, die Hauptsteuerung verwendet ESP32-PICO-D4-Chip, mit WIFI-Funktion, 135*240 Auflösung TFT-Bildschirm, eingebauter 160mAh Akku, seine „T-Lite“ Das T im Namen steht für Temperatur, und Lite steht für kleine, hoch integrierte.

- **M5StickC-PLUS:**

Abbildung 2-2: T-Lite M5StickC- plus [2]

Der M5StickC Plus ist eines der Kerngeräte der M5Stacks-Produktreihe. Das kompakte Gehäuse ist mit reichhaltigen Hardware-Ressourcen ausgestattet, wie Infra-rot, RTC, Mikrofon, LED, IMU, Tasten, PMU, etc. Verbesserungen gegenüber dem regulären StickC sind

Abbildung 2-3: M5StickC-Plus [3]

ein Summer, ein größerer Bildschirm (1,14-Zoll, 135 * 240 Auflösung LCD-Bildschirm) und ein stabileres Hardware-Design.

- **THERMAL HAT:**

THERMAL HAT ist eine M5StickC-kompatible Infrarot-Wärmebildkamera für Menschen. Eingebauter MLX90640 Thermosensor, der die Oberflächentemperatur eines Objekts misst und ein Wärmebild durch einen von der Oberflächentemperatur gebildeten Temperaturgradienten erzeugt. Die Auflösung beträgt (32 x 24).

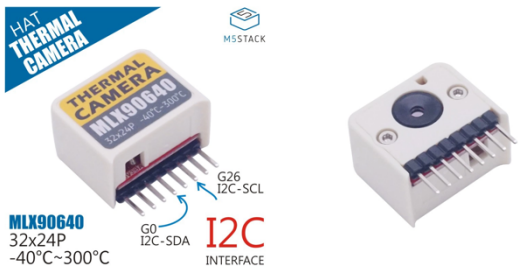


Abbildung 2-4: MLX90640 Thermosensor [8]

2.3 Master-Slave Prinzip:

In der digitalen Elektronik werden die Begriffe „Master“ und „Slave“ häufig verwendet, um die Beziehung zwischen zwei Geräten in einem Kommunikationsnetzwerk zu beschreiben. In unserem Fall ist der Mstack5 Core der Master und der Sensor MLX90640 ist der Slave.
In einer Master-Slave-Beziehung steuert das Master-Gerät die Kommunikation und initiiert die Datenübertragung, während das Slave-Gerät auf die Anfragen des Masters antwortet und Daten bereitstellt. Das Master-Gerät wird als verantwortlich betrachtet, während das Slave-Gerät, als derjenige betrachtet wird, der kontrolliert wird.
Die Master-Slave-Beziehung zwischen dem M5StickC Plus und dem MLX90640 Thermal HAT wird in der Regel durch ein I2C-Kommunikationsprotokoll (Inter-Integrated Circuit) ermöglicht. In dieser Konfiguration:

- Der M5StickC Plus initiiert die Kommunikation durch Senden eines Startsignals auf dem I2C-Bus.
- Der MLX90640 Thermal HAT wartet auf dem Bus auf Befehle vom Master und antwortet entsprechend.
- Der Datenaustausch erfolgt auf synchrone Weise, wobei der Master das Timing und den Fluss der Informationen kontrolliert.

3. Bild Interpolation Algorithmus:

In diesem Kapitel erklären wir die Interpolationstechniken, die wir in unserem Projekt verwendet haben, und wie wir sie implementiert haben. Anstatt direkt mit den Daten des THz-Sensors zu arbeiten, haben wir zunächst mit einem einfachen Bild begonnen. Dieses Bild hatte eine Auflösung von 32x24 Pixeln, und wir haben es in ein Array umgewandelt, um es dann zu interpolieren.
Unser Ziel war es, das Bild von 32x24 Pixeln auf 135x240 Pixel zu vergrößern. Dafür haben wir verschiedene Interpolationsmethoden verwendet, darunter:

3.1 Bilineare Interpolation:

Bilineare (2-D) Interpolation ist definiert als lineare Interpolation in zwei Richtungen oder Achsen. 1-D steht für eine Richtung (x-Achse), während 2-D für zwei Richtungen (x- und y-Achse) steht.

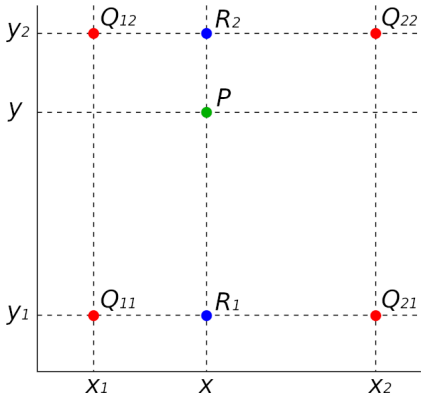


Abbildung 3-1: Bilineare Interpolation – Funktionsprinzip [4]

Wie funktioniert die bilineare (2-D) Interpolation? Nehmen wir an, dass wir einen Satz von Datenkoordinaten (xk, yk) definiert haben, wobei k = 1, 2. Diese Koordinaten bestimmen die Position der Punkte Q11, Q21, Q12 und Q22. Für beliebige x- und y-Koordinaten, die zwischen den Punkten xk und yk liegen, können wir durch Anwendung der bilinearen Interpolationstechnik den Punkt P (definiert durch x und y) finden.

- **Die formel:**

Die bilineare Interpolation besteht aus einer linearen Interpolation entlang beider Achsen, zwei linearen Interpolationen entlang der x-Achse und einer linearen Interpolation entlang der y-Achse.

- Der Punkt R1(x, y) ist definiert als:
  $R_1(x, y) = Q_{11} \cdot (x_2 - x) / (x_2 - x_1) + Q_{21} \cdot (x - x_1) / (x_2 - x_1)$
- Punkt R2(x, y) ist definiert als:
  $R_2(x, y) = Q_{12} \cdot (x_2 - x) / (x_2 - x_1) + Q_{22} \cdot (x - x_1) / (x_2 - x_1)$
- Der interpolierte Punkt P (x, y) ist definiert als:
  $P (x, y) = R1 \cdot (y2 - y) / (y2 - y1) + R2 \cdot (y - y1) / (y2 - y1)$

3.2 Bicubische Interpolation:

Bei der bicubischen Methode hingegen werden die Intensitätswerte mehrerer benachbarter Pixel berücksichtigt, um eine glattere Kurve zu erstellen, indem die zusätzlichen Informationen genutzt werden, um nicht lineare Funktionen zu erstellen, die das Auftreten von Unzulänglichkeiten, die bei der bilinearen Methode auftreten, verringern.

In 1d :

- Bilinear verwendet 2 Pixel zur Extrapolation des Pixelwerts.
- Bikubisch verwendet 4 Pixel für die Extrapolation des Pixelwerts.

In 2d :

- Bilinear verwendet 2 * 2 = 4 Pixel für die Extrapolation des Pixelwerts.
- Bikubisch verwendet 4 * 4 = 16 Pixel für die Extrapolation des Pixelwerts.

Daher ist die bikubische Interpolation für die Hochskalierung von Bildern rechenintensiver als die bilineare Interpolation, da mehr Werte als Zwischenschritte berechnet werden müssen.

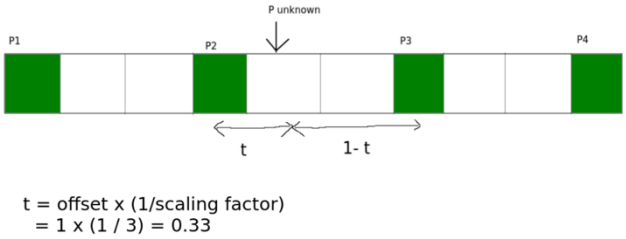


Abbildung 3-2: Interpolieren von Pixeln auf der gleichen horizontalen Linie wie bekannte Pixel [5]

- **Formeln:**

p = (benachbarte bekannte Punkte) und q = (Basisfunktion)
Die Formel für bicubisch sieht ähnlich aus wie die von Bilinear:

P (unknown) = p1q1 + p2q2 + p3q3 + p4q4

wobei p2 und p3 die Punkte sind, die dem unbekannten Punkt am nächsten liegen, und p1 und p4 andere Punkte in der Nähe von p2 bzw. p3 sind. Das Einzige, was sich ändert, ist die Basisfunktion.

- **$q1 = (-t^3 + 2t^2 — t) / 2$**
- **$q2 = (3t^3 — 5t^2 + 2) / 2$**
- **$q3 = (-3t^3 + 4t + t) / 2$**
- **$q4 = (t^3 — t^2) / 2$**

3.3 Nearest-Neighbour-Interpolation:

Dies ist die einfachste Form der Interpolation. Der Nearest Neighbor-Algorithmus berücksichtigt nur ein Pixel, nämlich das, das dem interpolierten Punkt am nächsten liegt. Dies erfordert von allen Interpolationsalgorithmen die geringste Verarbeitungszeit. Außerdem wird jedes Pixel einfach größer gemacht.

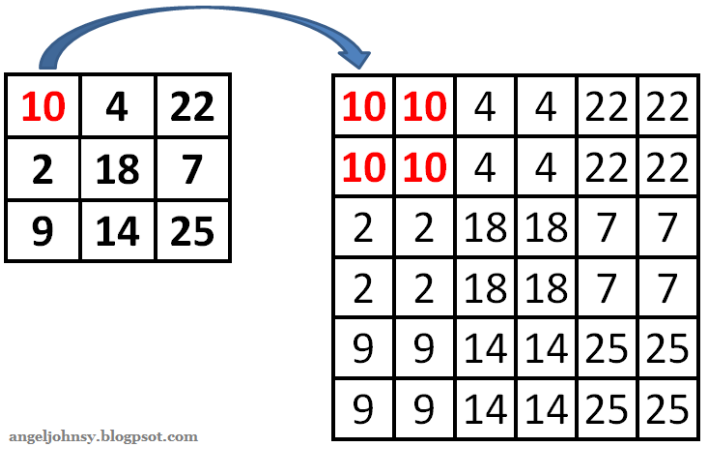


Abbildung 3-3: Nearest-Neighbor Interpolation Beispiel [6]

Die bildliche Darstellung zeigt, dass eine 3x3-Matrix zu einer 6x6-Matrix interpoliert wird. Die Werte in der interpolierten Matrix werden aus der Eingabematrix übernommen (d.h. es wird kein neuer Wert hinzugefügt).

Das Hauptmerkmal der Nearest Neighbour Interpolation ist, dass sie einfach den nächstgelegenen Pixelwert beim Vergrößern repliziert oder beim Verkleinern des Bildes reduziert. Diese Methode ist schnell und einfach zu implementieren, kann aber zu einem Verlust an Bildqualität führen, vor allem wenn Bilder stark vergrößert werden, da sie blockartige oder gezackte Kanten erzeugen kann. Daher wird sie für die Größenänderung hochwertiger Bilder oft nicht bevorzugt.

In bestimmten Situationen, in denen die Geschwindigkeit wichtiger ist als die Bildqualität, kann sie jedoch nützlich sein.

3.4 Isolierte Interpolationsmethode aus dem Original-Quellcode der T-LITE M5StickC Plus:

Neben den gängigen Interpolationsmethoden wie der linearen, bilinearen, bicubischen und der Nearest-Neighbour-Interpolation haben wir auch eine spezielle Interpolationsmethode implementiert, die wir aus dem Original-Quellcode der T-LITE M5StickC Plus isoliert haben. Dieser Algorithmus ist darauf ausgelegt, die Bilddaten durch Berücksichtigung der Temperaturänderungen zwischen benachbarten Pixeln zu interpolieren.

Der Algorithmus funktioniert so, dass er zunächst die Unterschiede in den Pixelwerten der umliegenden Bereiche berechnet, um Bereiche mit signifikanten Temperaturänderungen zu identifizieren. Anschließend erfolgt eine gewichtete Mittelung, bei der in Regionen mit geringen Temperaturänderungen die Werte aus dem vorherigen Bild stärker berücksichtigt werden. In Bereichen mit größeren Änderungen wird hingegen eine intensivere Interpolation durchgeführt. Diese Technik führt zu einem resultierenden Bild, das sowohl glatter als auch detailreicher ist.
Dieser spezielle Algorithmus kann im Original-Quellcode, speziell in der Datei **main.cpp** zwischen den Zeilen **4019** und **4120** gefunden werden.

4. Implementierung von Interpolationsalgorithmen:

4.1 Implementierung Von Bilinear:

```
#include "Bilnear.h"

#include <algorithm>
#include <thread>
#include <vector>

Bilnear::Bilnear() {}

Bilnear::~Bilnear() {}

// Diese Funktion führt die bilineare Interpolation für einen bestimmten
// Abschnitt des Bildes durch. Sie berechnet die neuen Pixelwerte basierend auf
// den Pixeln im Originalbild.

void interpolateSection(uint32_t oldWidth, uint32_t oldHeight,
                        uint8_t *oldPixels, uint32_t newWidth,
```

```
                        uint32_t newHeight, uint8_t *newPixels, float invScaleX,
                        float invScaleY, uint32_t startRow, uint32_t endRow) {
  // Diese Schleifen durchlaufen die Pixelzeilen (i) und Spalten (j) im neuen
  // Bildbereich. Dabei wird der entsprechende Quellpixelort im Originalbild
  // berechnet.
  for (uint32_t i = startRow; i < endRow; ++i) {
    uint32_t yi = i * newWidth * 3;
    float srcYFloat = (i * invScaleY);
    int srcY = static_cast<int>(srcYFloat);

    for (uint32_t j = 0; j < newWidth; ++j) {
      float srcXFloat = j * invScaleX;
      int srcX = static_cast<int>(srcXFloat);
      // Diese Bedingung stellt sicher, dass die berechneten Quellpixel
      // innerhalb der Grenzen des Originalbildes liegen. x_diff und y_diff
      // repräsentieren den Abstand zu den nächsten Pixeln.
      if (srcY >= 0 && srcY < oldHeight - 1 && srcX >= 0 &&
          srcX < oldWidth - 1) {
        float x_diff = srcXFloat - srcX;
        float y_diff = srcYFloat - srcY;
        // Berechnung der Indizes der vier benachbarten Pixel (A, B, C, D) im
        // Originalbild, die zur Berechnung des neuen Pixelwerts verwendet
        // werden.
        int indexA = (srcY * oldWidth + srcX) * 3;
        int indexB = indexA + 3;
        int indexC = ((srcY + 1) * oldWidth + srcX) * 3;
        int indexD = indexC + 3;
        // Diese Schleife interpoliert den neuen Farbwert (Rot, Grün, Blau)
        // basierend auf den Farbwerten der vier benachbarten Pixel im
        // Originalbild. Die neuen Werte werden in newPixels gespeichert.
        for (int k = 0; k < 3; ++k) {
          uint8_t A = oldPixels[indexA + k];
          uint8_t B = oldPixels[indexB + k];
          uint8_t C = oldPixels[indexC + k];
          uint8_t D = oldPixels[indexD + k];

          newPixels[yi + j * 3 + k] = static_cast<uint8_t>(
              A * (1 - x_diff) * (1 - y_diff) + B * x_diff * (1 - y_diff) +
              C * (1 - x_diff) * y_diff + D * x_diff * y_diff);
        }
        // Wenn der Quellpixelort außerhalb des Bildbereichs liegt, wird der
        // Farbwert des nächstgelegenen gültigen Pixels verwendet, um den neuen
        // Pixelwert zu setzen.
      } else {
        int edgeIndex = (srcY * oldWidth + srcX) * 3;
        for (int k = 0; k < 3; ++k) {
          newPixels[yi + j * 3 + k] = oldPixels[edgeIndex + k];
        }
      }
    }
  }
}
// Hauptfunktion zur Durchführung der bilinearen Interpolation. Sie skaliert das
// Bild von den ursprünglichen Abmessungen (oldWidth x oldHeight) auf die neuen
// Abmessungen (newWidth x newHeight).
// Rückgabewert: Ein Zeiger auf das neue, interpolierte Bild (newPixels).
uint8_t *Bilnear::interpolate(uint32_t oldWidth, uint32_t oldHeight,
                              uint8_t *oldPixels, uint32_t newWidth,
                              uint32_t newHeight) {
  // Berechnung der Skalierungsfaktoren für die Breite (scaleX) und Höhe
  // (scaleY) des Bildes.
  float scaleX = static_cast<float>(newWidth) / oldWidth;
  float scaleY = static_cast<float>(newHeight) / oldHeight;

  uint8_t *newPixels = new uint8_t[newWidth * newHeight * 3];

  float invScaleX = 1.0f / scaleX;
  float invScaleY = 1.0f / scaleY;
```

```
  // Ermittlung der Anzahl der Threads, die für die Interpolation verwendet
  // werden sollen, basierend auf der verfügbaren Hardware. rowsPerThread teilt
  // das Bild in Abschnitte, die parallel verarbeitet werden.

  unsigned int numThreads = std::min(std::thread::hardware_concurrency(), 8U);
  std::vector<std::thread> threads;
  uint32_t rowsPerThread = newHeight / numThreads;

  // Erstellung und Start der Threads, die jeweils einen Abschnitt des Bildes
  // interpolieren. Jeder Thread bearbeitet eine bestimmte Anzahl von Zeilen
  // (startRow bis endRow).
  for (unsigned int t = 0; t < numThreads; ++t) {
    uint32_t startRow = t * rowsPerThread;
    uint32_t endRow =
        (t == numThreads - 1) ? newHeight : (t + 1) * rowsPerThread;
    threads.emplace_back(interpolateSection, oldWidth, oldHeight, oldPixels,
                         newWidth, newHeight, newPixels, invScaleX, invScaleY,
                         startRow, endRow);
  }
  // Wartet darauf, dass alle gestarteten Threads ihre Arbeit beendet haben.
  for (auto &thread : threads) {
    thread.join();
  }
  // Gibt das neue Bild als Zeiger auf die interpolierten Pixel zurück.
  return newPixels;
}
```

4.2 Implementierung Von Bicubische Interpolation:

```
#include <cstdint>
#include <cstdio>

#include "BiubicInterpolator.hpp"

BiubicInterpolator::BiubicInterpolator(/* args */) {}

BiubicInterpolator::~BiubicInterpolator() {}

// Diese Methode führt die bikubische Interpolation durch, um die Größe eines
// Bildes zu ändern. Das Bild wird von den ursprünglichen Abmessungen (oldWidth
// x oldHeight) auf die neuen Abmessungen (newWidth x newHeight) skaliert.

// Rückgabewert: Ein Zeiger auf das neue, interpolierte Bild (newPixels).

uint8_t *BiubicInterpolator::interpolate(uint32_t oldWidth, uint32_t oldHeight,
                                         uint8_t *oldPixels, uint32_t newWidth,
                                         uint32_t newHeight) {
  // Berechnung der Skalierungsfaktoren für die Breite (scaleX) und Höhe
  // (scaleY) des Bildes.
  float scaleX = (float)newWidth / oldWidth;
  float scaleY = (float)newHeight / oldHeight;

  uint8_t *newPixels = new uint8_t[newWidth * newHeight * 3];

  // Diese Schleifen durchlaufen die Pixelzeilen (i) und Spalten (j) im neuen
  // Bildbereich und berechnen die entsprechenden Quellpixelkoordinaten (origX,
  // origY) im Originalbild. Die Schleife für i beginnt unten und arbeitet sich
  // nach oben, um die y-Achse anzupassen.

  for (int32_t i = newHeight - 1; i >= 0;
       i--) {  // Start from the lowest row and go up
    for (uint32_t j = 0; j < newWidth; j++) {
      int origX = (int)(j / scaleX);
```

```
      int origY =
          int origY = (int)(i / scaleY);// Adjust for inverted y-axis
      // Diese Bedingung stellt sicher, dass die Quellpixelkoordinaten im
      // gültigen Bereich des Originalbildes liegen. Die Funktion verwendet dann
      // bikubische Interpolation.
      if (origY >= 1 && origY < oldHeight - 2 && origX >= 1 &&
          origX < oldWidth - 2) {
        // Diese Schleifen berechnen den neuen Farbwert (Rot, Grün, Blau) für
        // jedes Pixel im neuen Bild durch bikubische Interpolation. Die
        // Farbwerte der umgebenden 16 Pixel im Originalbild werden verwendet,
        // um die neuen Farbwerte zu berechnen.
        // Die Methode cubicInterpolate() wird aufgerufen, um den bikubischen
        // Interpolationswert für die x- und y-Achsenverschiebungen (dx, dy) zu
        // berechnen.

        for (int k = 0; k < 3; k++) {
          float sum = 0.0;
          for (int m = -1; m <= 2; m++) {
            for (int n = -1; n <= 2; n++) {
              float dx = (j / scaleX) - origX;
              float dy = (i / scaleY) - origY; // Adjust for inverted y-axis
              float bx = cubicInterpolate(dx - n);
              float by = cubicInterpolate(dy - m);
              int pixelIndex = ((origY + m) * oldWidth + (origX + n)) * 3 + k;
              sum += bx * by * oldPixels[pixelIndex];
            }
          }

          int newPixelIndex = (i * newWidth + j) * 3 + k;
          newPixels[newPixelIndex] =
              sum > 255.0f ? 255 : (sum < 0.0f ? 0 : (uint8_t)sum);
        }
        // Wenn die Quellpixelkoordinaten außerhalb des gültigen Bereichs
        // liegen, wird der Farbwert des nächstgelegenen gültigen Pixels
        // verwendet, um den neuen Pixelwert zu setzen.
      } else {
        for (int k = 0; k < 3; k++) {
          int pixelIndex = (origY * oldWidth + origX) * 3 + k;
          int newPixelIndex = (i * newWidth + j) * 3 + k;
          newPixels[newPixelIndex] = oldPixels[pixelIndex];
        }
      }
    }
  }

  return newPixels;
}
// Diese Methode berechnet den bikubischen Interpolationswert basierend auf der
// gegebenen x-Verschiebung. Sie verwendet eine Formel, die auf dem Abstand vom
// Ursprungswert basiert.
float BiubicInterpolator::cubicInterpolate(float x) {
  // Berechnung des absoluten Wertes von x sowie seines Quadrats (absX2) und
  // seiner dritten Potenz (absX3).
  float absX = x < 0 ? -x : x;
  float absX2 = absX * absX;
  float absX3 = absX2 * absX;

  // Diese Bedingungen berechnen den interpolierten Wert basierend auf dem
  // Abstand x. Für absX ≤ 1 wird eine Formel verwendet, die den Wert für nähere
  // Pixel höher gewichtet. Für 1 < absX ≤ 2 wird eine andere Formel verwendet,
  // die den Einfluss von weiter entfernten Pixeln verringert. Wenn absX > 2,
  // wird 0.0f zurückgegeben, da die Pixel zu weit entfernt sind, um relevant zu
  // sein.
  if (absX <= 1.0f) {
    return (1.5f * absX3 - 2.5f * absX2 + 1.0f);
  } else if (absX <= 2.0f) {
    return (-0.5f * absX3 + 2.5f * absX2 - 4.0f * absX + 2.0f);
  }
```

```
  return 0.0f;
}
```

4.3 Implementierung Von Nearest-Neighbour Interpolation:

```
#include "NearestNeighbor.h"

NearestNeighbor::NearestNeighbor() {}

NearestNeighbor::~NearestNeighbor() {}

// Diese Methode führt die Interpolation eines Bildes mit der
// Nächster-Nachbar-Methode durch. Sie nimmt das Quellbild (src) und erzeugt
// eine vergrößerte oder verkleinerte Version (dst) basierend auf den neuen
// Abmessungen (dstWidth, dstHeight).
//
void NearestNeighbor::nearestNeighborInterpolate(uint8_t* src, int srcWidth,
                                                 int srcHeight, uint8_t* dst,
                                                 int dstWidth, int dstHeight) {
  // Diese Variablen berechnen die Skalierungsverhältnisse für die Breite
  // (xRatio) und Höhe (yRatio) zwischen dem Quell- und Zielbild.
  float xRatio = float(srcWidth) / dstWidth;
  float yRatio = float(srcHeight) / dstHeight;
  int nearestX, nearestY;

  // Diese Schleifen durchlaufen die Pixel des Zielbildes. Für jedes Pixel im
  // Zielbild wird das entsprechende Quellbildpixel (durch die
  // Nächster-Nachbar-Methode) ermittelt, indem die nächstgelegenen x- und
  // y-Koordinaten (nearestX, nearestY) im Quellbild berechnet werden.

  for (int i = 0; i < dstHeight; i++) {
    nearestY = int(i * yRatio);
    for (int j = 0; j < dstWidth; j++) {
      nearestX = int(j * xRatio);

      // Diese Abschnitte kopieren die RGB-Werte des nächstgelegenen Pixels im
      // Quellbild (src) in das Zielbild (dst). Der Index srcIndex wird für das
      // Quellbild und dstIndex für das Zielbild berechnet, und die
      // entsprechenden Farbwerte werden kopiert.
      int srcIndex = (nearestY * srcWidth + nearestX) * 3;
      int dstIndex = (i * dstWidth + j) * 3;
      dst[dstIndex] = src[srcIndex];
      dst[dstIndex + 1] = src[srcIndex + 1];
      dst[dstIndex + 2] = src[srcIndex + 2];
    }
  }
}
// Diese Methode dient als Schnittstelle zur Durchführung der
// Nächster-Nachbar-Interpolation. Sie erstellt ein neues Bild mit den
// angegebenen Abmessungen und verwendet die nearestNeighborInterpolate-Methode,
// um die Interpolation durchzuführen.

uint8_t* NearestNeighbor::interpolate(uint32_t oldWidth, uint32_t oldHeight,
                                      uint8_t* oldPixels, uint32_t newWidth,
                                      uint32_t newHeight) {
  // Allokiert Speicher für das neue Bild mit den spezifizierten Abmessungen
  // (newWidth x newHeight). Jedes Pixel besteht aus 3 Farbwerten (RGB).
  uint8_t* newPixels = new uint8_t[newWidth * newHeight * 3];
  // Führt die eigentliche Interpolation durch, indem die Methode
  // nearestNeighborInterpolate aufgerufen wird. Das neue Bild (newPixels) wird
  // basierend auf den Daten des ursprünglichen Bildes (oldPixels) erstellt.
  nearestNeighborInterpolate(oldPixels, oldWidth, oldHeight, newPixels,
                             newWidth, newHeight);
  return newPixels;
```

```
}
```

4.4 Implementierung Von Isolierte Interpolation von T-Lite Code Source:

```
#include "TemperatureInterpolator.h"

#include <cstddef>  // Fügt die Definition von size_t hinzu
#include <cstring>  // Für memcpy

TemperatureInterpolator::TemperatureInterpolator()
    : search_total(0),
      search_count(0),
      search_lowest(UINT32_MAX),
      search_highest(0) {}

TemperatureInterpolator::~TemperatureInterpolator() {}

uint8_t* TemperatureInterpolator::interpolate(uint32_t oldWidth,
                                              uint32_t oldHeight,
                                              uint8_t* oldPixels,
                                              uint32_t newWidth,
                                              uint32_t newHeight) {
  uint8_t* newPixels = new uint8_t[newWidth * newHeight * 3];  // RGB888 format (3
bytes per pixel)

  // Implementiere hier die Logik zur Anpassung von alten auf neue Dimensionen.

  // Calculate scaling factors
  float x_ratio = (float) oldWidth / newWidth;
  float y_ratio = (float) oldHeight / newHeight;

  for (uint32_t newY = 0; newY < newHeight; ++newY) {
    for (uint32_t newX = 0; newX < newWidth; ++newX) {
      // Map the new pixel position to the old pixel position
      uint32_t oldX = (uint32_t)(newX * x_ratio);
      uint32_t oldY = (uint32_t)(newY * y_ratio);

      // Ensure we do not go out of bounds
      oldX = oldX < oldWidth ? oldX : oldWidth - 1;
      oldY = oldY < oldHeight ? oldY : oldHeight - 1;

      // Calculate the index for old pixels
      size_t oldIndex =
          (oldY * oldWidth + oldX) * 3;  // Each pixel has 3 bytes (RGB)

      // Copy the pixel values from oldPixels to newPixels
      size_t newIndex = (newY * newWidth + newX) * 3;
      newPixels[newIndex] = oldPixels[oldIndex];          // R
      newPixels[newIndex + 1] = oldPixels[oldIndex + 1];  // G
      newPixels[newIndex + 2] = oldPixels[oldIndex + 2];  // B
    }
  }

  // Use interpolateFrame to adjust the pixel values based on temperature data
  uint8_t* diff =
      (uint8_t*)malloc(newWidth * newHeight);  // Example size; adjust as needed
  memset(diff, 0, newWidth * newHeight);       // Initialize diff array

  // Here, you would need the raw pixel data; using newPixels as a placeholder
  interpolateFrame((uint32_t*)newPixels, diff, newWidth, newHeight, 0, newWidth,
                   newHeight, 0, 0);

  // Hier könnte die tatsächliche Interpolationslogik eingefügt werden,
  // um die Temperaturdaten zu verarbeiten.
```

```
  free(diff);
  return newPixels;
}

void TemperatureInterpolator::interpolateFrame(uint32_t* pixel_raw,
                                               uint8_t* diff, int frame_width,
                                               int frame_height, int subpage,
                                               int mlx_width, int mlx_height,
                                               int monix, int moniy) {
    // Deine Interpolationslogik bleibt hier unverändert.
  for (int idx = 0; idx < 384; ++idx) {
    uint_fast8_t y = idx >> 4;
    uint_fast8_t x =
        ((mlx_width - 1 - (idx - (y << 4))) << 1) + ((y & 1) != subpage);
    uint_fast16_t xy = x + y * frame_width;

    uint32_t diff_sum = 0;
    size_t count = 0;
    if (x > 0) {
      ++count;
      diff_sum += diff[(xy - 1) >> 1];
    }
    if (x < (frame_width - 1)) {
      ++count;
      diff_sum += diff[(xy + 1) >> 1];
    }
    if (y > 0) {
      ++count;
      diff_sum += diff[(xy - frame_width) >> 1];
    }
    if (y < (frame_height - 1)) {
      ++count;
      diff_sum += diff[(xy + frame_width) >> 1];
    }
    diff_sum /= count;

    int32_t raw = pixel_raw[xy];

    uint32_t sum = 0;
    if (x > 0) {
      sum += pixel_raw[xy - 1];
    }
    if (x < (frame_width - 1)) {
      sum += pixel_raw[xy + 1];
    }
    if (y > 0) {
      sum += pixel_raw[xy - frame_width];
    }
    if (y < (frame_height - 1)) {
      sum += pixel_raw[xy + frame_width];
    }
    raw = (sum + (count >> 1)) / count;

    if (diff_sum > 256) {
      diff_sum = 256;
    }
    raw = (pixel_raw[xy] * (256 - diff_sum) + diff_sum * raw) >> 8;
    pixel_raw[xy] = raw;

    updateMinMaxTemperature(pixel_raw, x, y, frame_width, frame_height,
                            mlx_width, mlx_height, monix, moniy);
  }
}

void TemperatureInterpolator::updateMinMaxTemperature(
    uint32_t* pixel_raw, int x, int y, int frame_width, int frame_height,
    int mlx_width, int mlx_height, int monix, int moniy) {
  uint_fast16_t xy = x + y * frame_width;
```

```
  if (((moniy + y - (mlx_height >> 1)) < (moniy << 1)) &&
      ((monix + x - (mlx_width)) < (monix << 1))) {
    search_total += pixel_raw[xy];
    ++search_count;
    if (search_lowest > pixel_raw[xy]) {
      search_lowest = pixel_raw[xy];
    }
    if (search_highest < pixel_raw[xy]) {
      search_highest = pixel_raw[xy];
    }
  }
}
```

4.5 Implementierung Von Main.cpp :

```
// Diese Header-Dateien sind erforderlich, um die Funktionen und Klassen des
// Codes zu nutzen. Dazu gehören Bibliotheken für das M5Core2-Display,
// Bildverarbeitung und verschiedene Interpolationsmethoden.

#include <M5Core2.h>
#include <image.h>

#include <cstdint>

#include "Bilnear.h"
#include "BiubicInterpolator.hpp"
#include "LinearInterpolator.h"
#include "NearestNeighbor.h"
#include "OptimizedBilinearInterpolator.h"
#include "TemperatureInterpolator.h"
#include "bilinearInterpolationNew.h"

/* Definiert maximale Displaybreite und -höhe sowie die ursprünglichen und neuen
 * Bildabmessungen. Diese Werte werden für die Interpolation und Anzeige
 * verwendet.*/

#define MAX_DISP_WIDTH 240
#define MAX_DISP_HEIGHT 135

const int origWidth = 32;
const int origHeight = 24;

const int displayWidth = 240;
const int displayHeight = 135;

TFT_eSprite img = TFT_eSprite(&M5.Lcd);

TemperatureInterpolator interpolator;  // Using the TemperatureInterpolator

void rgb888_to_rgb565_row(uint8_t *rgb888Row, uint16_t *rgb565Row, int width);

int rotationSetting = 1;  // Set rotation: 0 (0°), 1 (90°), 2 (180°), 3 (270°)

// Die setup-Funktion initialisiert die serielle Kommunikation, das M5-LCD und
// das Sprite. Der Bildschirm wird mit schwarzer Farbe gefüllt, die Farbentiefe
// auf 16 Bit gesetzt und die Rotation angewendet.
void setup() {
  Serial.begin(115200);
  M5.begin();
  M5.Lcd.fillScreen(TFT_BLACK);
  img.setColorDepth(16);
  M5.Lcd.setRotation(rotationSetting);  // Apply the desired rotation
  img.createSprite(displayWidth, displayHeight);
}
```

```
uint_fast64_t startTime;
uint_fast64_t endTime;

void loop() {
  startTime = millis();  // Hier wird die aktuelle Zeit in Millisekunden
                         // gespeichert, um die Leistung zu messen.

  img.fillSprite(TFT_BLACK);

  // Ruft die interpolate-Methode des TemperatureInterpolator-Objekts auf, um
  // das ursprüngliche Bild (imagePixels) auf die neuen Abmessungen zu
  // interpolieren. Das Ergebnis wird in newpixels gespeichert.

  uint8_t *newpixels = interpolator.interpolate(
      origWidth, origHeight, imagePixels, displayWidth, displayHeight);

  //Überprüft, ob die Interpolation erfolgreich war. Wenn nicht, wird eine
  // Fehlermeldung auf dem Sprite angezeigt, und die Funktion wird beendet.
  if (newpixels == nullptr) {
    img.printf("Interpolation failed!");
    return;
  }

  // Buffer for a single row conversion from RGB888 to RGB565
  uint16_t *rowPixels565 = new uint16_t[displayWidth];

  //Überprüft die erfolgreiche Speicherzuweisung für rowPixels565. Wenn die
  // Zuweisung fehlschlägt, wird eine Fehlermeldung angezeigt, und der zuvor
  // allokierte Speicher für newpixels wird freigegeben, um einen Speicherleck
  // zu vermeiden.
  if (rowPixels565 == nullptr) {
    img.printf("Memory allocation for rowPixels565 failed!");
    delete[] newpixels;  // Clean up to avoid memory leak
    return;
  }

  // Convert and push each row separately
  // Diese Schleife konvertiert jede Zeile des neuen Bildes (newpixels) von
  // RGB888 in RGB565 und zeigt sie auf dem Sprite an. Die Zeilen werden von
  // unten nach oben angezeigt, um die Koordinaten des M5-Displays zu
  // berücksichtigen.
  for (int y = 0; y < displayHeight; y++) {
    rgb888_to_rgb565_row(newpixels + y * displayWidth * 3, rowPixels565,
                         displayWidth);
    img.pushImage(0, displayHeight - y - 1, displayWidth, 1, rowPixels565);
  }

  endTime = millis();
  img.setCursor(10, 10);
  img.setTextColor(TFT_RED);
  img.printf("FPS: %f", 1000.0 / (endTime - startTime));
  img.pushSprite(0, 0);

  delete[] newpixels;
  delete[] rowPixels565;
}


// Optimized function for row-by-row RGB888 to RGB565 conversion
void rgb888_to_rgb565_row(uint8_t *rgb888Row, uint16_t *rgb565Row, int width) {
  for (int i = 0; i < width; i++) {
    int index = i * 3;  // Index for RGB888

    // Get RGB components
    uint8_t b = rgb888Row[index];
    uint8_t g = rgb888Row[index + 1];
    uint8_t r = rgb888Row[index + 2];
```

```
    // Convert to RGB565
    rgb565Row[i] = img.color565(r, g, b);
  }
}
```

Wir haben verschiedene Interpolationsverfahren implementiert und diese anschließend in die Geräte M5Core2 und M5StickC Plus integriert. Dabei sind wir jedoch auf einige Herausforderungen gestoßen.

- **Umwandlung und Inversion von Bitmap-Daten:**

Wir haben das Bild im „**Bitmap**“-Format, wobei die erste Zeile tatsächlich die letzte im Daten-Array von Bytes ist. Daher müssen wir die Zeilen invertieren. Zusätzlich speichert die Bitmap jeder Pixel in 3 Bytes (B, G und R) im RGB888-Format. Um diese Daten in ein 24-Bit-Farbformat (RGB565) umzuwandeln, verwenden wir eine Funktion namens **void(rgb888_to_rgb565),** die beide Aufgaben ausführt.

- **Das Speicherproblem:**

Zunächst haben wir beim Konvertieren von RGB888 nach RGB565 Speicher für das gesamte Bild zugewiesen, was jedoch nicht möglich war, da der verfügbare Speicher nicht ausreichte. Daher bestand die Lösung darin, nur eine einzelne Zeile zuzuweisen, diese anzuzeigen und dann zur nächsten Zeile überzugehen, um sie zu konvertieren und ebenfalls anzuzeigen, und so weiter.

Abschließend haben wir die Leistung der Algorithmen auf diesen Geräten getestet, indem wir die Bildwiederholrate (FPS) und die Effizienz der Implementierungen miteinander verglichen haben.

- Diese Projekte werden auf Github hochgeladen:

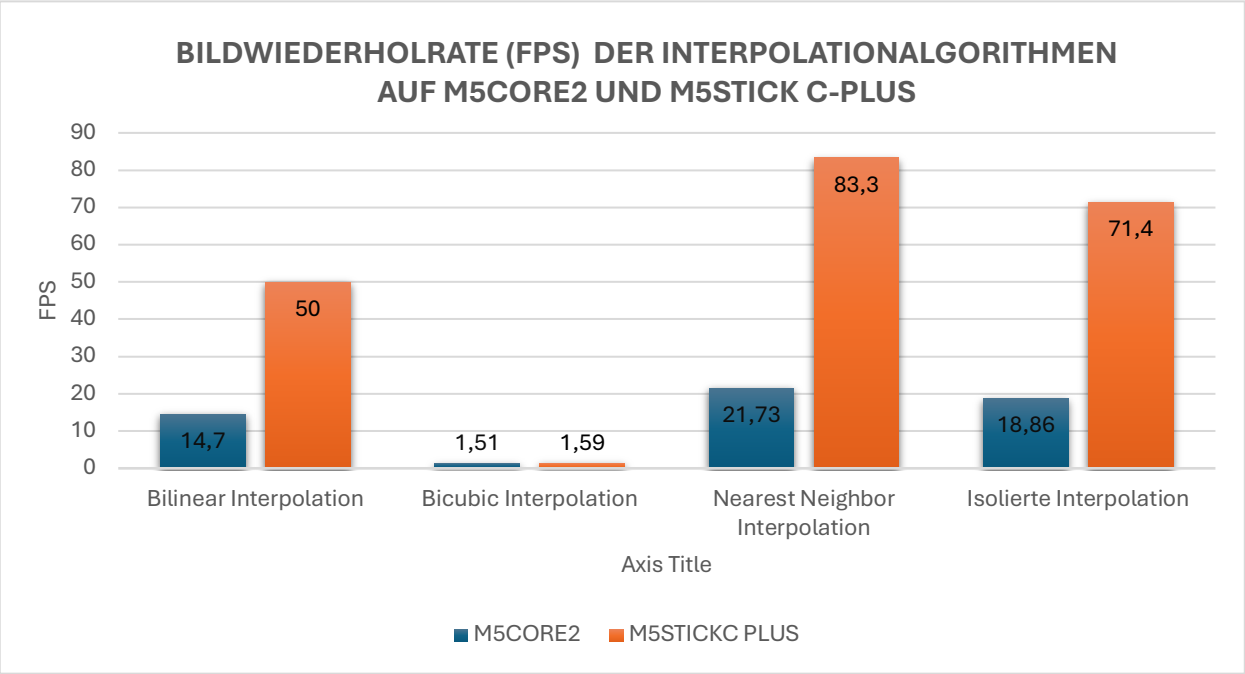
Implementierung für M5CORE2 : saifel96/THzVideoCameraInterpolation-M5Core2 (github.com)

Implementierung für M5Stick-c-plus : saifel96/THzVideoCameraInterpolation-M5StickCPLUS (github.com)

5. Data Visualisieren und Ergebnisse:

Um einen fairen Vergleich zwischen den beiden Hardwareplattformen, M5Core2 und M5StickC Plus, zu ermöglichen, haben wir unsere Interpolationsalgorithmen auf beiden Geräten getestet. Dabei verwendeten wir eine Bildgröße von 135x240 Pixeln für alle Verfahren. Die folgende Tabelle zeigt die Unterschiede in der Bildwiederholrate (FPS) für die verschiedenen Algorithmen auf den beiden Geräten:

	M5CORE2	M5STICKC PLUS
Bilinear Interpolation	14.70 (FPS)	50 (FPS)
Bicubic Interpolation	1.51 (FPS)	1.59 (FPS)
Nearest-Neighbour Interpolation	21.73 (FPS)	83.3 (FPS)
Isolierte Interpolation von Orignal Code Source	18.86 (FPS)	71.4 (FPS)



Diese Ergebnisse zeigen deutliche Unterschiede in der Performance zwischen den beiden Geräten. Während M5StickC Plus in den meisten Fällen eine höhere Bildwiederholrate erreicht, insbesondere bei der bilinearen und Nearest-Neighbour-Interpolation, zeigt die bicubische Interpolation auf beiden Geräten ähnliche und niedrigere Werte. Die aus dem Original-Quellcode isolierte Interpolation liefert ebenfalls eine signifikant bessere Leistung auf dem M5StickC Plus im Vergleich zum M5Core2.

6. Analyse der Leistungsunterschiede zwischen M5Core2 und M5StickC Plus:

Obwohl wir die gleiche Software auf beiden Geräten, dem M5Core2 und dem M5StickC Plus, verwendet haben, gibt es einen wesentlichen Unterschied in der Leistung. Dieser Unterschied ist nicht auf die Software zurückzuführen, sondern lässt uns über andere mögliche Ursachen nachdenken, die hardwareabhängig sind.
Die Unterschiede in der Leistung (FPS) zwischen den beiden Geräten, M5Core2 und M5StickC Plus, können aus mehreren Gründen entstehen:

6.1 Prozessorarchitektur und Unterschiede zwischen den ESP32-Chips:

Beide Geräte verwenden den ESP32-Chip, aber es gibt subtile Unterschiede zwischen den Varianten. Der ESP32 D0WDQ6-V3 im M5Core2 und der ESP32-PICO im M5StickC Plus sind zwar ähnliche Chips, aber der ESP32-PICO ist eine kompaktere Version, die alle notwendigen Komponenten (wie Flash-Speicher, Quarz und Antenne) auf einem einzigen Modul integriert. Dies kann den Stromverbrauch und die Effizienz verbessern, was zu einer besseren Leistung bei Interpolationsaufgaben führen könnte.

6.2 Integriertes Crystal (Quarz):

Der ESP32-PICO enthält einen integrierten Quarzkristall, der eine präzise Taktfrequenz bereitstellt. Hier sind einige mögliche Auswirkungen auf die Leistung:

Ein integriertes Crystal bedeutet, dass die Taktung des Prozessors genau abgestimmt und stabil ist. Es gibt weniger externe Faktoren, die die Taktfrequenz beeinflussen können (wie elektrische Störungen oder Temperaturschwankungen). Diese genaue Taktung könnte dazu führen, dass der ESP32-PICO konstant mit der maximal möglichen Geschwindigkeit arbeitet, was eine höhere Effizienz und bessere Leistung bei Prozessen wie der Bildverarbeitung und Interpolation ermöglicht.

6.3 Integrierter 4 MB Flash-Speicher:

Der integrierte 4 MB Flash-Speicher des ESP32-PICO kann ebenfalls die Leistung bei Bildverarbeitungsaufgaben erheblich beeinflussen. Hier sind die Gründe, warum dies der Fall sein könnte:

Da der Flash-Speicher direkt im ESP32-PICO integriert ist, kann die CPU schneller auf den Speicher zugreifen. Im Vergleich zu einem externen Flash-Speicher (wie möglicherweise beim M5Core2), bei dem zusätzliche Latenzen beim Speicherzugriff auftreten können, arbeitet der ESP32-PICO effizienter, da weniger Zeit für den Datenaustausch zwischen CPU und Speicher benötigt wird. Dies ist besonders wichtig, wenn große Datenmengen wie Bilddaten verarbeitet werden müssen.

6.4 Weniger externe Komponenten:

Da der ESP32-PICO viele seiner Komponenten, einschließlich Crystal und Flash-Speicher, intern integriert hat, gibt es weniger externe Verbindungen und damit weniger Signalverluste oder Latenzen. Externe Verbindungen, wie sie im M5Core2 möglicherweise existieren, könnten die Verarbeitungsgeschwindigkeit beeinträchtigen. Je weniger externe Komponenten ein Gerät hat, desto direkter und schneller kann die CPU auf wichtige Ressourcen wie den Flash-Speicher zugreifen.

7. Zusammenfassung:

Im Rahmen dieses Projekts haben wir uns mit der Bildverarbeitung und Interpolation von Thermalkameradaten auf zwei verschiedenen Geräten, dem M5Core2 und dem M5StickC Plus, beschäftigt. Ziel war es, Bilddaten einer THz-Kamera von einer Auflösung von 32x24 Pixeln auf 135x240 Pixel zu interpolieren und verschiedene Interpolationsmethoden zu testen. Zu den angewandten Algorithmen gehörten lineare, bilineare, bicubische und Nearest-Neighbour-Interpolation sowie eine spezielle Interpolation, die wir aus dem Quellcode des M5StickC Plus isolieren konnten.

Um die Effizienz dieser Algorithmen zu evaluieren, haben wir sie auf beiden Geräten implementiert und getestet. Dabei wurden die FPS (Frames per Second) als Leistungskennzahl herangezogen, um die Geschwindigkeit der Bildverarbeitung zu vergleichen. Die Ergebnisse zeigten signifikante Unterschiede zwischen den beiden Geräten, wobei der M5StickC Plus in den meisten Fällen höhere FPS erzielte als der M5Core2.

Die Unterschiede in der Leistung lassen sich vor allem auf hardwaretechnische Faktoren zurückführen. Obwohl auf beiden Geräten die gleiche Software ausgeführt wurde, spielten die Unterschiede in der Prozessorarchitektur, der Taktfrequenzstabilität durch integrierte Quarze und die Effizienz des integrierten Flash-Speichers des ESP32-PICO-D4 eine entscheidende Rolle für die bessere Performance des M5StickC Plus.

Insgesamt konnte durch dieses Projekt gezeigt werden, wie wichtig die Hardwarekonfiguration für die Performance von Bildverarbeitungsalgorithmen ist. Die Ergebnisse liefern wertvolle Erkenntnisse für die weitere Optimierung von Bildverarbeitungsprozessen auf Embedded-Systemen, insbesondere im Hinblick auf Echtzeit-Anwendungen wie die Thz-Video Camera.

8. LiteraturVerzeichnis:

[1]. M5Stack Core2. [Online]. M5Stack Core2 ESP32 IoT Development Kit for AWS IoT Kit | m5stack-store

[2]. T-Lite. [Online]. m5-docs (m5stack.com)

[3]. M5StickC PLUS. [Online]. m5-docs (m5stack.com)

[4]. Bilinear interpolation. [Online]. Bilinear interpolation – x-engineer.org.

[5]. Image Upscaling using Bicubic Interpolation. [Online]. Image Upscaling using Bicubic Interpolation | by Amanrao | Medium. (Abrufdatum: 23. September 2023)

[6]. Nearest Neighbor Interpolation .[Online].Nearest Neighbor Interpolation | IMAGE PROCESSING (imageeprocessing.com)

[7]. M5StickC-Plus-TLite-FW. [Online]. m5stack/M5StickC-Plus-TLite-FW: M5StickT-Lite internal firmware (M5StickC-Plus + MLX90640 HAT) (github.com)

[8]. THERMAL HAT. [Online]. m5-docs (m5stack.com).