Abkürzungsverzeichnis

NVCC NVidia Cuda Compiler

1 Detektor & Filter

Bilder werden ueber ein Kameramodul aufgenommen, einem internen neuronalen Netzwerk uebergeben und auf erkannte Objekte (Fahrzeuge) ausgewertet. Sind Objekte — oder auch keine Objekte — erkannt worden, findet eine Informationsweitergabe statt. Die Ergebnisse des Neuronalen Netzwerks und seinem unterliegendem Algorithmus (YoLo) werden der Verarbeitungspipeline uebergeben.

Hintergedanken fuer diese Herangehensweisse:

- Weiterfuehrende Datenaufbereitungen sind somit von "Detektor" Prozess entkoppelt
  - Neue Algorithmen koennen sehr einfach durch andere Algorithmen der "Daten-Pipeline" ersetzt oder hinzugefuegt werden.
  - Abarbeitungsreihenfolgen der Algorithmen koennen beliebig neu angeordnet werden
  - Neue Algorithmen koennen in gewuenschter Sprache implementiert und der Pipeline hinzugefuegt werden
    - Somit keine Neukompilierung von eiem einzigen *monolitischen* Prozesses mit NVidia Cuda Compiler (NVCC) notwendig

Ein digitlaer Filter ist ein diskreter Algorithmus, eine Rechenvorschrift. Wenn hier von einem Filter gesprochen wird, ist ein Algorithmus in einem eigenen Ausfuehrungsprozess gemeint. Werden mehrere Filter hintereinander ausgefuhrt, findet eine Pipeline-Verarbeitung statt[1]. Es existiert ein unidirektionaler Kommunikationskanal zwischen den Prozessen, d. h. die Kommunikationsrichtung fliesst nur in eine richtung.

Fragen, die hier behandelt werden, sind:

- Wie sind die Prozesse intern aufgebaut
- Wie sind deren Schnittstellen zur *Aussenwelt* definiert

Detector

Die Architektur ist Stufenweise aufgebaut. In **??** ist zu sehen, Softwareteile hoeherer Ordnung sind weiter oben angeordnet und deren Abhaengigkeiten liegen tiefer und Verbindungslinien kennzeichnen die Abhaengigkeiten.

Der oberste Block kennzeichnet den Hauptprozess. Die naechsten beiden Softwareblocke sind der *Command Line Parse* und die *Detect Funktion*. Der Command Line Parser wird

[1] https://www.elektronik-kompendium.de/sites/com/1705221.htm

beim ersten Prozessstart ausgefuehrt und liesst die in der Kommandozeile beim Starten des Prozess definierten Zusatzparameter und dekodiert diese, was die weitere Prozessabarbeitung beeinflusst. Bspw. wird der Prozess wie folgt:

```
$ detector --visual-mode
```

gestartet, liesst der Parser das uebergebene Flag und die Ausgabe des Prozesses ist fuer den Anwender in lesbarer Form. Im *visual mode* ist aber eine Prozessausfuehrung innerhalb der Pipeline so nicht moeglich. Der ausgehende Prozess muss Schnittstellenkonform mit dem nachfolgenden Prozess der Pipeline sein, was *visual mode* erfuellt.

Die *Detect Funktion* ist der in Matlab generierte CUDA Algorithmus fuer die Objekterkennung. Der Algorithmus wird in einer Endlosschleife zyklisch ausgefuehrt. Die direkten Abhaengikeiten des Algorithmus sind die CUDA Libraries. Vorteil von Matlab generierten Algorithmus ist: Softwareentwickler definiert den Algorithmus in Matlab-Code und Matlab produziert den CUDA Code, kompiliert ihn und erzeugt eine statische Library, die in ein unabhaengiges Projekt eingefuegt werden kann. Wie hier: Eingliederung CUDA Library in Detector Hauptprozess. Somit muss der Entwickler nicht direkt mit den CUDA Libraries arbeiten. Deren Aufrufe sind in der statischen Library *wegabstrahiert*. Ergebnis ist: Es kann weitgehend auf CUDA Code Syntax verzichtet werden und mit C++ respektive C Code Style gearbeitet werden, weil eine gemeinsame Schnittmenge mit CUDA Syntax vorhanden ist.

Der Hauptfunktionalitaet ist ein *Signal Handler* uebergeordnet. Der Signal Handler reagiert auf System Signale und fuer diese, wofuer er definiert wurde, unterbricht er den Hauptprozess und stellt sicher, dass die abhaengige Hardware wie Kamera Modul sicher gelschlossen wird. Bspw. mit dem asynchronen Befehl:

```
$ SIGINT
```

wird der Prozess unterbrochen und die Hardware und diverse Filedeskriptoren geschlossen. Vor dem Signal Handler traten Probleme auf bei Testdurchlaeufen, in denen das Programm mehrmals unterbrochen wurde, dass die Hardware nicht mehr reagierte. Die Hauptfunktionalitaet uebergibt gewonnenen Daten dem *Output-Stream*.

Informationen werden mittels dem Betriebssystem bereitgestellten Pipelinesystem uebertragen. Die Schnittstellendefinition zum naechsten Prozess der Pipeline ist daher jeweil der Outputstream. Dem uebergeordneten Verarbeitungsprozess (Filter) wird mit einem definierten Signal mitgeteilt, das ab diesem Zeitpunkt gueltige Daten am Stream anliegen. Ein Codeausschnitt ist:

```
// 1ter Prozess der Pipeline
std::out << some_random_data;
my::flag(std::cout, 0xBADEAFFE);
std::out << valid_data;

////////////////////////////////

// 2ter Prozess der Pipeline
do_some_random_stuff();
wait(my::flag(std::cin, 0xBADEAFFE));
do_important_stuff();
```

Bash Skript starte bspw. beide Prozesse in einer Pipelinestruktur:

```
$ Detector | Filter1
```

Gibt *Detector* Daten vor dem Flag *0xBADEAFFE* aus, werden diese von *Filter1* nicht weiter bearbeitet. Erst wenn das Flag gelesen wurde, werden die nachfolgenden Daten des Streams als gueltige Daten betrachtet und gehen in die Bearbeitung mit ein.

Bleibt nur noch zu klaeren, welche Daten der *Detector* Prozess der Pipeline uebergibt.

Literaturverzeichnis

[Nvi01a] Nvidia:
JETSON NANO.
http://www.nvidia.com/de-de/autonomous-machines/embedded-systems/jetson-nano/education-projects/, 2020-09-01. –
Last Accessed: 2021-04-14

[Nvi01b] Nvidia:
JETSON NANO - Getting Started.
http://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit, 2020-09-01. –
Last Accessed: 2021-04-14