Vorlesung Architekturen und Entwurf von Rechnersystemen

Prof. Andreas Koch, Yannick Lavan, Johannes Wirth, Mihaela Damian

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 2022/2023
Übungsblatt 3

In dieser und der folgenden Übung betrachten wir die Implementierung eines Beschleunigers für Gaussfilter und Software-gestützte Testbenches in BSV.

Aufgabe 3.1: Gaussfilter Teil 1

Gaussfilter sind in der Bildverarbeitung einfache Faltungsfilter, die eine Glättung des Eingabebilds bewirken, also Kanten abschwächen. In dieser und den nächsten Übungen entwickeln wir einen Beschleuniger für 3×3 Gaussfilter.

3.1a) Einleitung

Ein Gaussfilter[1] verwendet die Dichte der Normalverteilung, um ein Signal zu glätten. Wir werden eine Approximation implementieren, die eine 3×3 Faltungsmatrix mit dem Bild faltet. Eine Faltung ist hierbei eine gewichtete Summe eines 3×3 Bildausschnitts. Die Gewichte sind hierbei die Zahlen in der Faltungsmatrix. Das Konzept wird in Abbildung 1 illustriert.

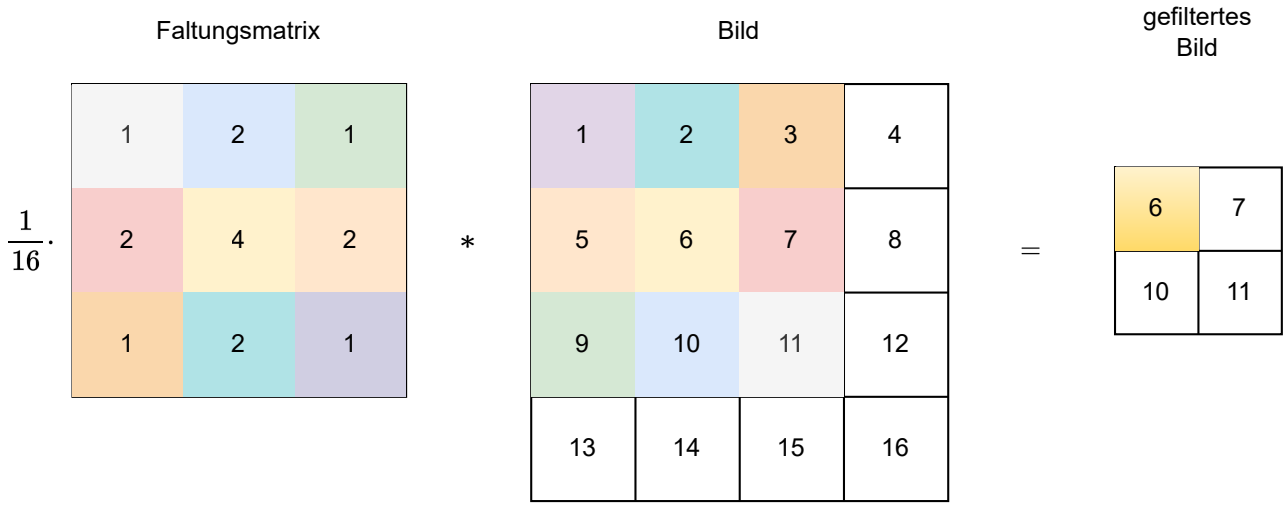

Abbildung 1: Jeder Pixel des aktuellen Ausschnitts wird mit dem passenden Eintrag in der Faltungsmatrix multipliziert, die Ergebnisse werden aufsummiert und anschließend durch 16 dividiert.

[1] https://en.wikipedia.org/wiki/Gaussian_blur

Für die Faltung eines 3×3 Bildes g mit der Faltungsmatrix 3×3 k gilt also:

$$(k * g)(0,0) = \begin{bmatrix} a & b & c \\ d & e & f \\ h & i & j \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = 9a + 8b + 7c + 6d + 5e + 4f + 3h + 2i + 1j \quad (1)$$

$a - j$ sind hierbei die Gewichte der Faltungsmatrix.

Wir ignorieren die Randfälle, in denen nicht genügend Pixel für eine vollständige Berechnung des Filters vorhanden sind und arbeiten nur auf Pixeln, die vollständig von Pixeln des Bildes umgeben sind. Dadurch wird das Bild kleiner. Das Bild g aus Gleichung 1 enthält also nur einen Ergebnispixel.

Abbildung 2 veranschaulicht den Aufbau des Beschleunigers. Wir erläutern kurz die Funktionalität:

Graustufenpixel kommen sequentiell im Beschleuniger an, im besten Fall mit einem neuen Pixel pro Takt. Jedes mal wenn ein neuer Pixel ankommt wird ein Wert von einem Register zum nächsten geschoben.

Da die meisten Pixel mehrfach benötigt werden, werden sie in Puffern (`mkBuffer`) zwischengespeichert. Beispielsweise wird der Pixel mit dem Wert 10 in Abbildung 1 für die Berechnung der Pixel 6, 7, 10 und 11 verwendet. Hätte das Eingabebild eine weitere Zeile würde er außerdem noch für die Berechnung der Pixel 14 und 15 verwendet werden.

Für ein höheres Maß an Abstraktion, werden die beiden Puffer in einem übergeordneten Modul (`mkRowBuffer`) eingebaut. Diese Puffer beginnen erst mit der Weitergabe von Pixeln, wenn sie eine Reihe des Bildes angenommen haben (mehr dazu in der zugehörigen Teilaufgabe).

Wenn das Arbeitsfenster vollständig besetzt wurde erfolgt im nächsten Takt die Weitergabe an die eigentliche Berechnungseinheit (`mkGauss`). Von dort aus wird dann die Output-FIFO mit den Ergebnissen befüllt.

In dieser Übung werden Sie die Grundkomponenten implementieren. In der nächsten Übung setzen Sie dann die Komponenten zum Gesamtsystem zusammen.

Wir stellen in Moodle ein Template zur Verfügung:

- `Buffer.bsv` - Package für Aufgabe 3.1
- `BufferTb.bsv` - Package für Aufgabe 3.1
- `CImg.h` - Bibliothek[2] für die Verwendung von Bildverarbeitungsfunktionen in Testbenches
- `Gauss.bsv` - Package für Aufgabe 3.1
- `GaussChecker.bsv` - Package für Aufgabe 3.1
- `ImageFunctions.bsv` - Package mit Hilfsfunktionen und CImg-Wrappern
- `ImageReader/ImageWriter.cpp` - C++ Quelldateien für CImg-Wrapper
- `Makefile` - Makefile zum Bauen und Testen Ihrer Implementierungen
- `MyTypes.bsv` - Package für zusätzlich definierte Typen
- `oracle.cpp` - C++ Quelldatei für Orakelwrapper aus Aufgabe 3.1
- `picture.png` - Testbild
- `RowBuffer.bsv` - Package für Aufgabe 3.1
- `Settings.bsv` - Package für Konfigurationsparameter, z.B. Bildauflösung

[2] Bibliothek: https://cimg.eu/index.html, Lizenz: http://www.cecill.info/licences/Licence_CeCILL-C_V1-en.txt

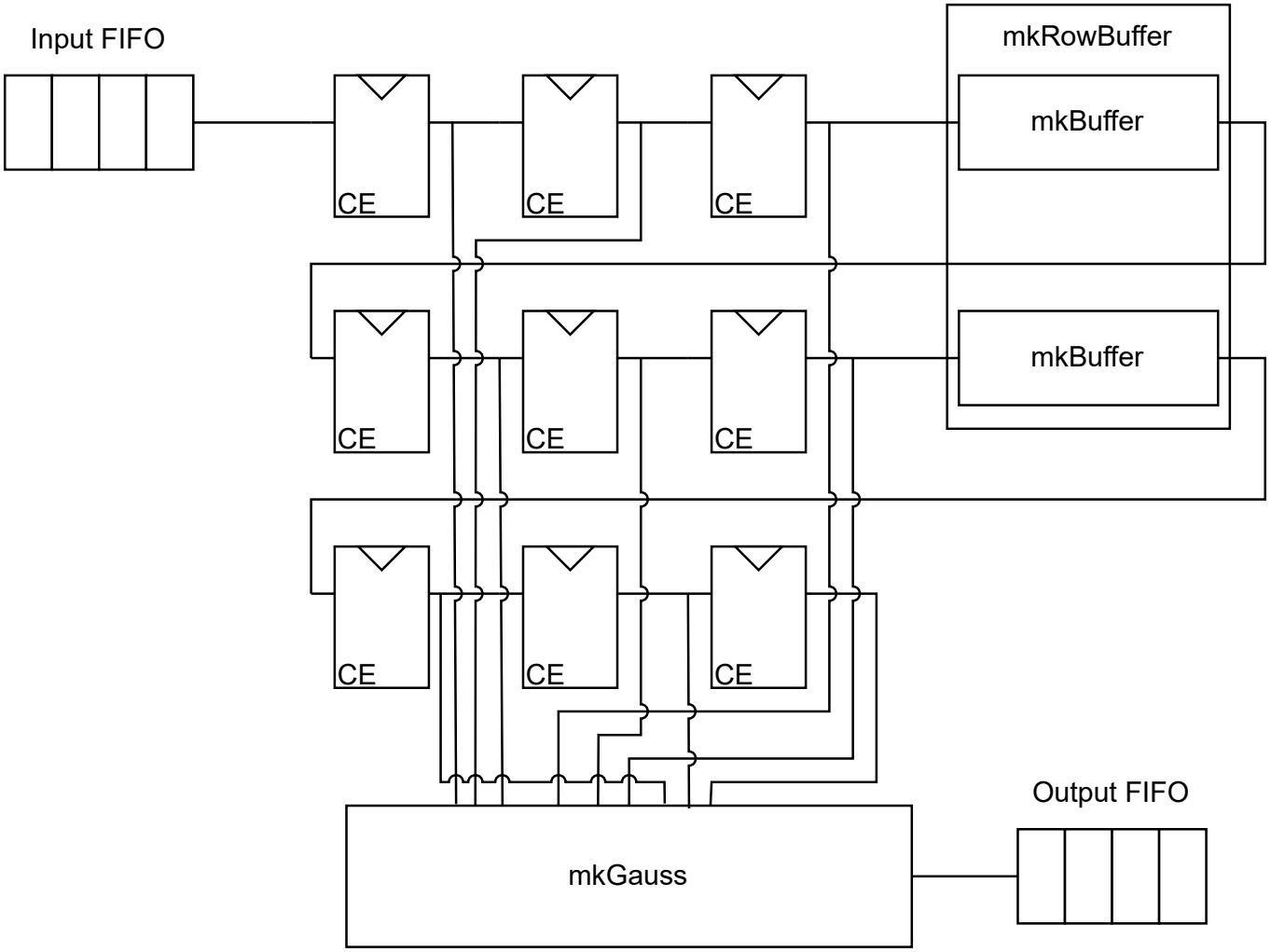

Abbildung 2: Stream Architektur des Beschleunigers. Pixel erreichen den Beschleuniger nacheinander. Das Arbeitsfenster wird in einem Registerfeld gespeichert. Erneut benötigte Pixel werden zwischengespeichert.

3.1b) Gaussfilter Implementierung

Implementieren Sie das Modul `mkGauss` (Datei `Gauss.bsv`). Das Modul soll ein `Server`-Interface implementieren, welches als Request einen `Vector` aus neun Graustufenpixeln (Typ `GrayScale`) annimmt und als Response das Ergebnis der Faltung vom Typ `GrayScale` zurückliefert. Ankommende Requests sollen in einer `FIFO` gespeichert werden, ausgehende Responses sollen in einer anderen `FIFO` gespeichert werden. Die Faltungsmatrix soll „fest verdrahtet“ sein und die Werte aus Abbildung 1 beinhalten.

Einige Fragen, die Sie sich vor der Implementierung stellen sollten:

1. Finden die Multiplikationen parallel im gleichen Takt oder sequenziell in mehreren Takten statt?
2. Welche Bitbreiten werden für Zwischenergebnisse benötigt?
3. Fällt Ihnen etwas an der Faltungsmatrix auf?

Sie können Ihre Implementierung mit dem Befehl `make bluesim_GaussTb` bauen und anschließend mit `./GaussTb_bluesim` testen.

Sollten Sie den Filter mit einem anderen Bild testen wollen, so können Sie den Dateinamen in `GaussTb.bsv` Z. 50 ändern und die Auflösung in `Settings.bsv` anpassen. Sollten Sie eine ähnliche Fehlermeldung wie die folgende erhalten, so können Sie den Wert des Flags `-D_GLIBCXX_USE_CXX11_ABI=1` ausprobieren.

```
Error: dlopen: ./GaussTb_bluesim.so: undefined symbol: _Z8copy_argRKSs
    invoked from within
"sim load $model_name $top_module"
    (file "/opt/bsc-2022.01-ubuntu-20.04/lib/tcllib/bluespec/bluesim.tcl" line 188)
```

3.1c) Gausskernel - Selbstprüfende Testbench

Die in der vorherigen Teilaufgabe genutzte Testbench liefert Ihnen eine Möglichkeit die Ergebnisse Ihres Filters visuell zu inspizieren und zu debuggen. In einem Projektkontext ist diese Art von Testbench aber nicht ideal, da sie nicht wirklich automatisiert verwendet werden kann und regelmäßige Tests hierdurch erschwert werden. Andererseits ist es mühsam eine vollständige Referenzimplementierung in einer HDL (BSV, SV, VHDL, etc.) zu implementieren.

Dafür bieten HDLs oft die Möglichkeit C-Code zu importieren und innerhalb von Testbenches zu verwenden. Diese Funktionalität erlaubt es uns ein Orakel in C(++) zu implementieren und es in unserer Testbench für Referenzwerte zu nutzen.

Wir geben Ihnen bereits eine Referenzimplementierung in C vor und haben Ihnen die entsprechenden Funktionen in `ImageFunctions.bsv` importiert. Dort finden Sie auch weitere nützliche Funktionen. Nutzen Sie diese Funktionen, um eine *selbstprüfende* Testbench zu implementieren. Vervollständigen Sie hierfür das Modul `mkGaussChecker` in `GaussChecker.bsv`.

Sie können Ihre Implementierung mit

```
make bluesim_GaussChecker
./GaussChecker_bluesim
```

testen. Sie finden Beispiele für die Nutzung der Bildfunktionen in `GaussTb.bsv`.

3.1d) Pufferimplementierung

Im nächsten Schritt implementieren wir die Zwischenspeicher, die immer aktuell nicht benötigte Pixel einer aktuellen Reihe speichern. Implementieren Sie dafür zunächst das Modul `mkBuffer` (in `Buffer.bsv`). Das Modul soll ein `Server`-Interface implementieren, dessen Request und Response Typ jeweils ein `Maybe#(GrayScale)` ist. Eingabewerte sollen in einer Input-FIFO ankommen, von dort in einer `mkSizedBRAMFIFO` gespeichert werden und aus dieser wieder in eine Output-FIFO geschoben werden, sobald genug Pixel angekommen sind. Eine Skizze des Designs finden Sie in Abbildung 3. Wir verwenden `Maybe`, um das Scheduling im `RowBuffer` zu vereinfachen.

Sie müssen noch weitere Kontrolllogik einführen, da ein ankommender Wert ansonsten direkt von `bufferedValue` zur Ausgangs-FIFO `outputValue` weitergereicht wird.

Überlegen Sie sich zunächst wie viele Pixel ein Buffer abhängig von der Bildbreite halten muss. Überlegen Sie sich anschließend welche konkrete FIFO-Implementierung Sie für die Input- und Output-FIFO verwenden.

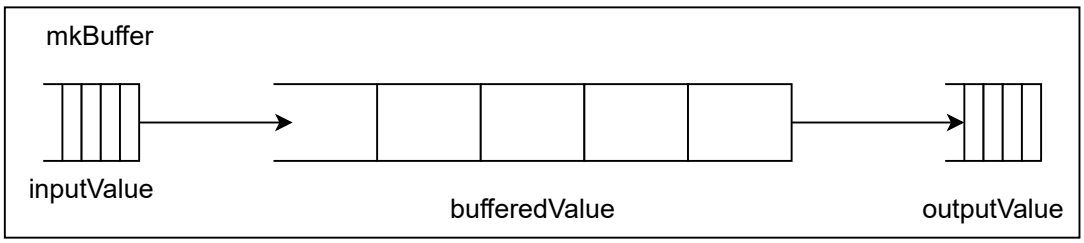

Abbildung 3: Skizze des Buffer-Moduls. Neben den gezeigten FIFOs wird noch Kontrolllogik benötigt.

3.1e) Buffer Testbench

Implementieren Sie eine selbstprüfende Testbench, die das funktionale Verhalten des Buffers testet. Rufen Sie sich die funktionalen Anforderungen des Buffers ins Gedächtnis:

1. Er kann mindestens x Werte halten (wobei x aus der ersten Frage der vorherigen Aufgabe resultiert).
2. Er beginnt frühestens dann mit der Ausgabe neuer Werte wenn x Werte eingegeben wurden.

3.1f) RowBuffer Implementierung

Implementieren Sie nun das Modul `mkRowBuffer` (in `RowBuffer.bsv`). Es soll beide benötigten Buffer beinhalten und ein `Server`-Interface mit Request- und Responsetyp `Vector#(2, Maybe#(GrayScale))` implementieren.

Designfrage: Müssen die beiden Buffer innerhalb des RowBuffers Werte unabhängig voneinander ausgeben können?