Fachprojekt:
RISC-V mit Matrixmultiplikation Erweiterung

23.03.2022

Inhaltsverzeichnis

1 Einleitung/Motivation

In diesem Repository wurde ein Matrixbeschleuniger mit einer Risc-V CPU verbunden, um NN schneller ausführen zu können.

In der Regel braucht ein Multiplikationswerk mehr Zeit, um ein Ergebnis auszurechnen als ein Additionswerk. Um eine Matrixmultiplikation auszuführen, bräuchte ein CPU viel Zeit, wenn sie keine speziellen Matrixfunktionen besitzt. Programme, die mit NN arbeiten, müssen aber häufig Matrixmultiplikationen ausrechnen. Mit Hilfe dieses Projekts wollen wir Programme, die auf NN beruhen, beschleunigen. Die CPU, die wir für das Projekt ausgewählt haben, kann Teile des 32 Bits Risc-V Instruktionsets ausführen und der Beschleuniger erwartet Matrizen, dessen Werte auf 8 Bit quantisiert sind (-128 bis 127). Die beiden Komponenten haben wir über ein 32-Bit Interface miteinander verbunden.

1.1 Grundidee

Die Grundidee von Beschleunigern ist es eine langwierige Operationen an den Beschleuniger auszulagern. In unserem Fall -ein Matrixmultiplikationsbeschleuniger- muss der Beschleuniger als erstes die Dimensionen der Matrizen kennen. Der Beschleuniger kann dabei nur mit zweidimensionalen Matrizen umgehen. Die ersten Daten/32 Bit müssen dabei wie folgt aufgebaut sein:

- Bit 31 – 24: Anzahl an Zeilen in der ersten Matrix
- Bit 23 – 16: Anzahl an Spalten in der ersten Matrix
- Bit 15 – 08: Anzahl an Zeilen in der zweiten Matrix
- Bit 07 – 00: Anzahl an Spalten in der zweiten Matrix

Danach muss der Inhalt der beiden Matrizen übertragen werden. Der Beschleuniger erwartet dabei, dass erst Matrix Eins und dann Matrix Zwei übertragen wird. Beiden werden zeilenweise übertragen, also: a_{11}, a_{12}, ..., a_{1n}, ..., a_{mn}. Da der Beschleuniger mit 8 Bit Werten arbeitet, über das Interface aber 32 Bit werte übertragen werden, sind bei der Datenübertragen Bit 31 – 08 egal, nur Bits 07 – 00 sind wichtig. Die Übertragung für die Multiplikation von zwei 2×2 Matrizen sähe wie folgt aus:

1. X“02020202“ Übertragung der Dimensionen
2. X“——“ & a_{11} Beginn von der Übertragung der ersten Matrix
3. X“——“ & a_{12}
4. X“——“ & a_{21}
5. X“——“ & a_{22}
6. X“——“ & b_{11} Beginn von der Übertragung der zweiten Matrix
7. X“——“ & b_{12}
8. X“——“ & b_{21}
9. X“——“ & b_{22}

X“——“ beschreibt die ersten drei Bytes, dessen Wert nicht relevant ist.

Nachdem der Beschleuniger mit der Berechnung fertig ist, schreibt dieser die Ergebnismatrix (zeilenweise) zurück an das Interface (die ersten drei Bytes werden mit Nullen gefüllt).

2 Gatter

2.1 Interface

Damit der Beschleuniger und die CPU mit dem kleinst möglichen Taktzyklus arbeitet, wurde entschieden, dass die beide Module unterschiedliche Taktgeber haben. Das Interface ist eine Art von Buffer, um die Daten den Taktdomänen zu synchronisieren. Als Input braucht das Interface beide Takte, sowie jeweils einen Dateneingang (32 Bit) und ein Steuersignal, ob geschrieben werden soll, und ein Steuersignal, ob gelesen werden soll, pro Taktdomäne.

Als Output gibt es die beiden Datenausgänge (32 Bit), sowie die Kontrollsignale, ob geschrieben und gelesen werden kann.

Im Interface befinden sich zwei Ring-Buffer (für jede Richtung Einen), die die Funktionalitäten ausüben.

2.1.1 Ring-Buffer

Intern besitzt der Ring-Buffer zwei Pointer, der eine zeigt auf die nächste Speicherzelle, die beschrieben wird, der zweite zeigt auf die nächste Speicherzelle, aus der gelesen werden soll. Der Ring-Buffer hat zwei Takteingänge, der eine ist mit der Schreiboperation synchron, der andere ist mit der Leseoperation synchron.

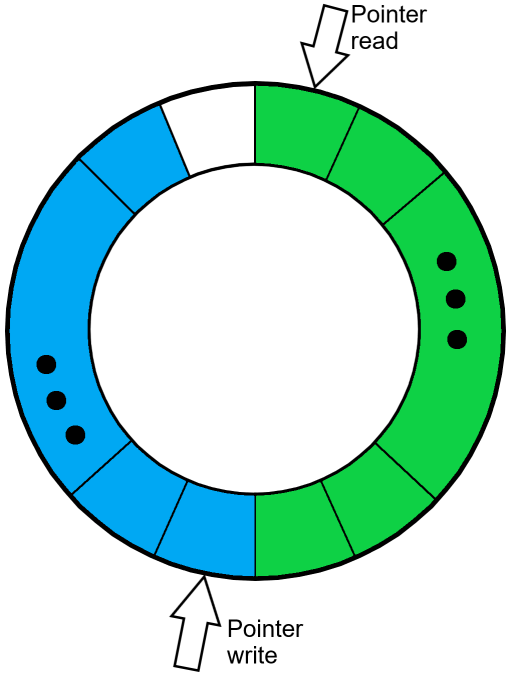


Abbildung 1:
Grün stellt im Bild den Bereich dar, den man lesen kann.
Blau stellt im Bild den Bereich dar, den man beschreiben kann.
Weiß stellt im Bild den Bereich dar, den man weder lesen noch beschreiben kann.

Der weiße Bereich existiert wegen der Implementierung. Es wurden folgende Annahmen getroffen bei der Implementierung, wann man in den Buffer schreiben darf und wann man lesen darf:

1. Wenn der Schreib bzw. Lese Befehl kommt wird an der Stelle, an der der Pointer gerade ist, gelesen bzw. geschrieben.
2. Der Schreib-Pointer ist immer vor oder auf der gleiche Position, wie der Lese-Pointer
3. Kein Pointer darf den anderen Pointer überholen.

4. Der Schreib-Pointer darf sich selbst nicht auf die Position vom Lese-Pointer bewegen
5. Die Werte der Pointer befinden sich zwischen 0 und size-1

Dadurch entstehen folgenden Bedingungen wann man lesen/schreiben darf:

- Schreiben (wenn 1x erfüllt darf man schreiben)
  - adr_write < size-1 and adr_write > adr_read
  - adr_write = size-1 and adr_read > 0
  - adr_write = adr_read
  - adr_write < adr_read-1
- Lesen
  - adr_read /= adr_write

adr_write/read = Pointer für write/read, size = Größe des Buffers
Damit das Schreiben einen gesamten Zyklus zu schreiben bekommt wird adr-read nicht direkt mit adr_write verglichen, sondern mit adr_write_old, welches adr_write ist nur einen Takt verzögert.

Mit den Ports write_data, read_data wird dem Buffer gesagt, ob in diesem Takt geschrieben/gelesen werden soll. can_write, can_read gibt aus, ob in diesem Takt geschrieben bzw. gelesen werden kann.

Dadurch hat man das Problem der Synchronisation in gewisser Weise verschoben, den die Pointer adr_write und adr_read werden in unterschiedliche Takten geschrieben/verändert und gelesen. Um Fehler bei der Synchronisation zu vermeiden, werden die Pointer mit Hilfe des Grey-Codes in die andere Taktdomäne übertragen. Dadurch, das beim Grey-Code sich nur ein Bit ändern darf, kann der Pointer nur zwei mögliche Werte annehmen, die beide zu einem korrekten Ergebnis kommen. [5]

2.2 Beschleuniger

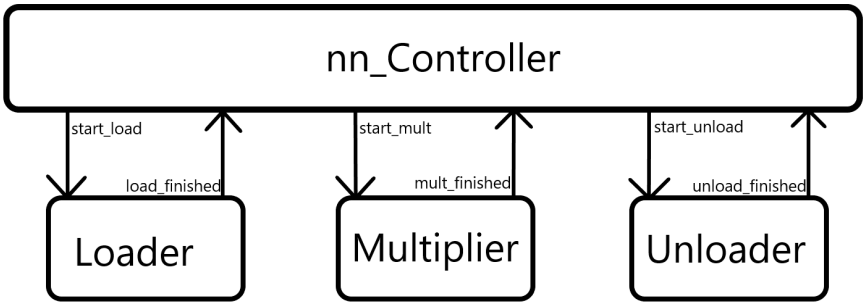
Der Beschleuniger durchläuft vier verschiedene Phase.

1. Auf Daten warten
2. Daten laden
3. Matrixmultiplikation berechnen
4. Daten zurückschreiben

Im Beschleuniger gibt es deswegen, 4 Controller. Der erste übernimmt das Laden der Daten, der zweite führt die Matrixmultiplikation aus, der dritte schreibt die Daten zurück und der letzte Controller kontrolliert die Controller, sagt ihnen wann sie aktiv werden dürfen und wann sie warten sollen.

2.2.1 nn_Controller

Der nn_Controller ist der Controller, der die Controller kontrolliert. Der nn_Controller wartet auf einen Input im Interface, ob eines bereit liegt sieht dieser durch den Port can_read. Wenn das erste Datum vorliegt berechnet der nn_Controller die Größen der Matrizen. Diese gibt er durch die Outputs rows1, columns1, rows2 und columns2 aus. Den Rest steuern die Untercontroller. Sobald alle Untercontroller ihre Aufgabe erledigt haben, wartet der Controller wieder auf einen neuen Wert im Interface. Die Unterroutine werden gestartet durch die Outputs: start_load, start_mult und start_unload. Wenn diese Signal 1 sind, wird den Untercontroller, gesagt das die entsprechende Operationen nun starten müssen. Das Signal bleibt auf dem Wert 1 bis der entsprechende Input: load_finished, mult_finished, unload_finished den Wert 1 erhält. Dann weiß der nn_Controller, das der entsprechende Untercontroller seine Operation abgeschlossen hat und der Controller kann in die nächste Phase übergehen.



2.2.2 Load-Controller

Der Controller ist dafür da, die Daten in einen Matrixspeicher zu laden. Dafür erhält die Größe der Matrizen vom nn_Controller und weiß die bereits die Größe des Beschleunigers. Der Load-Controller sagt dann an welche Adresse das Datum hin muss (Output: pos_x11, pos_x12, pos_x21, pos_x22) und sagt in welchen Matrixspeicher es soll, durch die Outputs write_enable und 2.

Die Matrizenspeicher werden auf spezielle Art beschrieben, wegen des Multiplikationsverfahren. Die erste Matrix wird soweit oben und rechts in den Matrixspeicher geschrieben wie es geht. Die zweite Matrix hingegen wird soweit nach unten und links wie es geht.

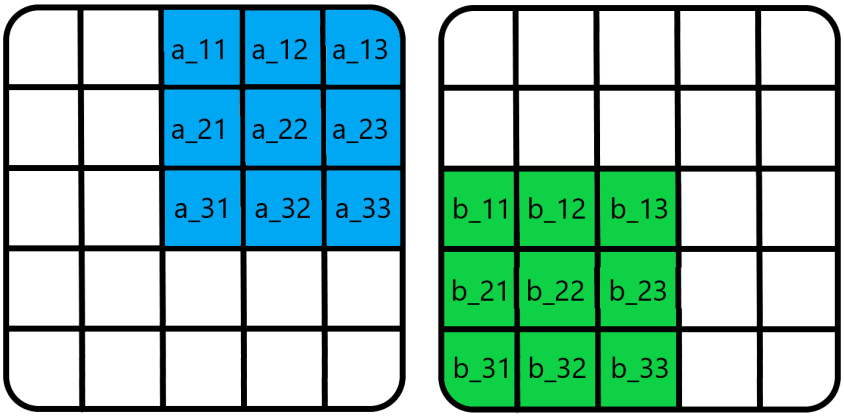


Abbildung 2: Zwei 3x3 Matrizen in 5x5 Speicherzellen:

2.2.3 fifo_grid/fifo_grid_vertical

fifo_grid und fifo_grid_vertical sind Speicher in denen Matrizen gespeichert werden. Beide haben einen synchronen Reset. Beide schreiben an die Position von pos_x1 und pos_x2, wenn write_enable = 1 und eine positive Taktflanke vorliegt. Beide habe eine shift-Funktion, die ausgelöst wird wenn shift = 1 ist, aber sie schieben in unterschiedliche Richtungen. fifo_grid schiebt in horizontale Richtung, wohin gegen fifo_grid_vertical in vertikaler Richtung verschiebt (siehe Abbildung 3).

Die shift-Funktion steht für den Multiplikation-Controller zu Verfügung.

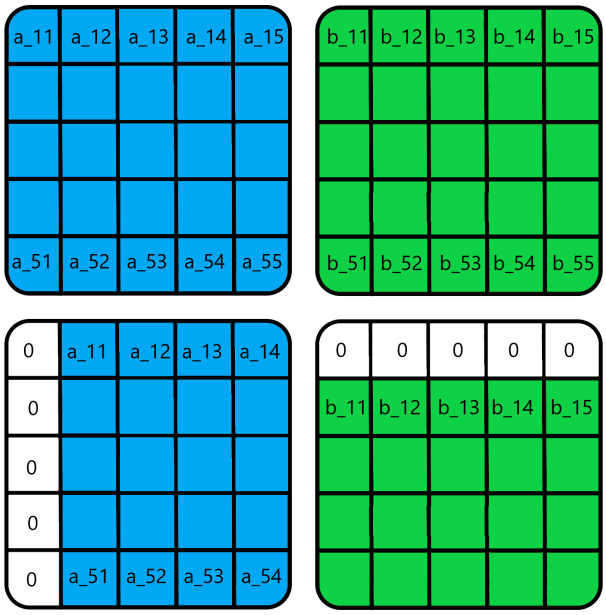


Abbildung 3: Links sieht man fifo_grid, rechts sieht man fifo_grid_vertical.

2.2.4 Multiplikation-Controller

Die Multiplikation wird von einer anderen Gruppe ausgeführt.
https://github.com/killgrabber/FP6-Computation-Unit

2.2.5 Unload-Controller

Der Controller ist dafür da, das Ergebnis in das Interface zu schreiben. Dafür erhält die Größe der Ergebnismatrix vom nn_Controller und die Ergebnismatrix von dem Multiplizierer. Der Unload-Controller berechnet die Adresse für das nächste Datum in der Ergebnismatrix, ließt dann das Datum aus und schreibt es dann in das Interface, wenn der Buffer im Interface noch nicht voll ist, dies erfährt der Controller durch den Input can-write.

2.2.6 NN

Das Gatter nn ist der komplette Beschleuniger. Es enthält die oben genannten Gatter, die entsprechend mit einander verbunden sind.

2.3 CPU

Die Gatter in der CPU wurden leicht angepasst, um mit dem Beschleuniger und dem Interface umgehen zu können.

2.3.1 Hazzardunit

Durch die neuen Instruktionen zum senden und empfangen von Daten aus dem Interface, entstehen auch neue Pipeline Hazzards.

Daten an das Interface zu senden, passiert in der Write-Back Phase der Pipeline. Daten vom Interface werden in der execute Phase empfangen.

Dabei entstehen nur Hazzards, wenn man schreiben/lesen und der Buffer vom Interface voll/leer ist. Sollte man lesen wollen, aber man kann nicht muss man die ersten drei Stufen der Pipeline stallen, sowie

die vierte Stufe blockieren mit einem flush. Sollte man schreiben wollen, aber der Buffer ist voll, wird die gesamte Pipeline gestallt bis man wieder schreiben darf.

2.3.2 ALU

Die Alu kann nun zwei weitere Operationen ausführen. Nämlich slli und eine acht Bit Addition. Slli ist die normale shift-left Operation, die in den Risc-V Spezifikationen vorkommt. Die acht Bit Addition nimmt die unteren 8 Bits von den Inputs SrcA und SrcB und interpretiert diese im Zweierkomplement Format. Falls ein Over/Underflow passiert, wird das Ergebnis auf den maximalen positiven/negativen Wert gesetzt (127/-128).

2.3.3 Weitere Instruktionen

Da in den Matrizen 8 Bit Werte gespeichert sind, enthält die CPU zusätzlich noch Byte load und store Instruktionen. Load Byte funktioniert sowie lw nur, indem die oberen 3 Bytes mit Nullen gefüllt werden und nur das untere Byte vom Register einen Wert aus dem Speicher erhält. Store Byte funktioniert sowie sw nur, indem die oberen 3 Bytes nicht in den Speicher geschrieben werden, sondern nur das unterste Byte. Um diese zu realisieren wurden weitere Multiplexer in die CPU eingesetzt.

	31 30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0
store Byte (storeB)	Imm[11:5]	rs2 = content	rs1 = Adress	Value not needed	Imm[4:0]	1 1 1 0 1 0 0
load Byte (loadB)	Imm[11:0]		rs1 = Adress	Value not needed	rd	1 1 1 0 0 0 0
toAccelerator (toAcc)	Imm[11:0]		rs1 = Adress	Value not needed	Value not needed	1 1 0 0 0 0 0
toAcceleratorByte (toAccB)	Imm[11:0]		rs1 = Adress	Value not needed	Value not needed	1 1 0 0 1 0 0
fromAccelerator (fromAcc)	Imm[11:5]	Value not needed	rs1 = Adress	Value not needed	Imm[4:0]	1 1 0 1 0 0 0
formAcceleratorByte (fromAccB)	Imm[11:5]	Value not needed	rs1 = Adress	Value not needed	Imm[4:0]	1 1 0 1 1 0 0
add Byte (addB)	Value not needed	rs2	rs1	0 1 1	rd	1 1 1 1 0 0 0

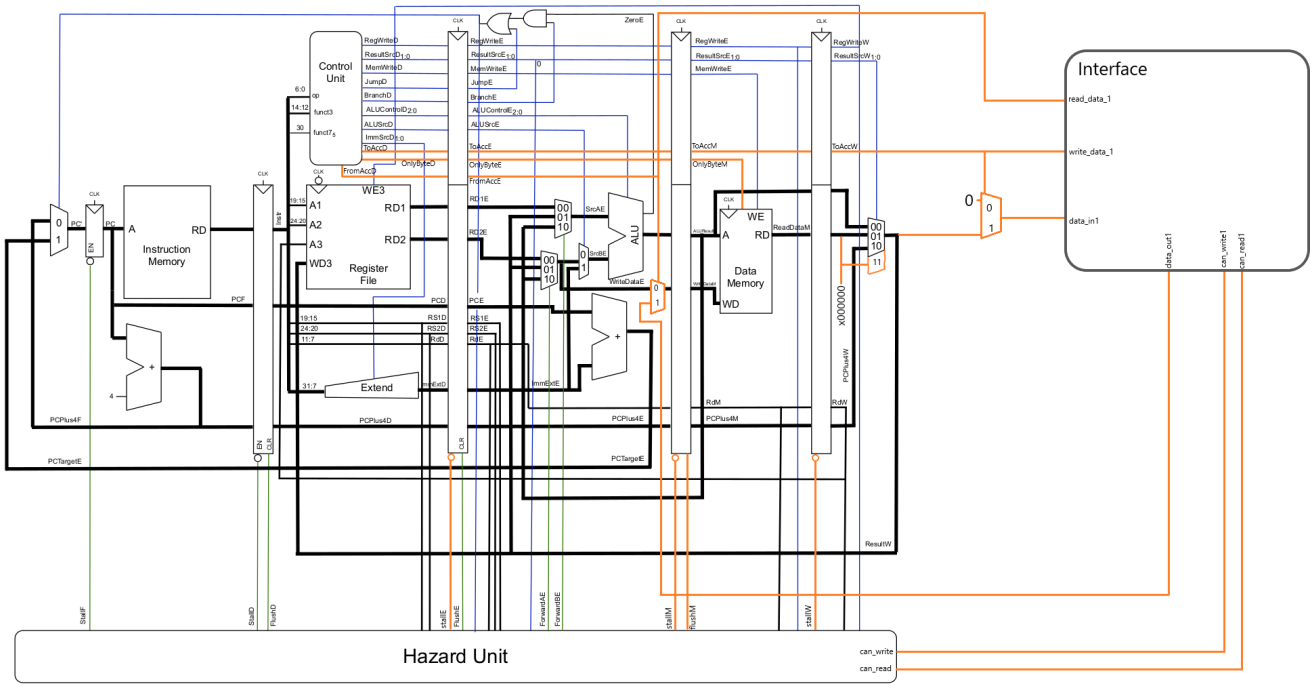


Abbildung 4: Die Orangenlinien wurden hinzugefügt.

2.4 Assembly-Code

In dem Repository sollte auch noch ein Programm liegen, das verschiedene Neuronale Netze ausführen kann. Dieses Programm ist zeit technischen Gründen nicht fertig geworden. Viele nötige Funktionen sind nicht implementiert.
Der Plan war ein Program in Assembly-Code zu schreiben und diesen dann mit Hilfe eines Skriptes den Assembly-Code in die Datei instr_mem.vhdl zu schreiben.
Das Programm würde viel mit Matrizen umgehen müssen. Matrizen, die zu groß für den Beschleuniger wären, müssten aufgeteilt werden in sogenannte nicht zusammenhängende Matrizen. Die Teile des Programmes, die existieren, speichern Matrizen auf folgende Weise ab:

- Zusammenhängende Matrix:
  - Die ersten 32-Bit beschreiben die Anzahl an Reihen.
  - Die zweiten 32-Bit beschreiben die Anzahl an Spalten.
  - Danach kommen in 8-Bit Schritten, die Inhalte der Matrix (zeilenweise).
- Nicht zusammenhängende Matrix:
  - Die ersten 32-Bit beschreiben die Anzahl an Reihen-Matrizen.
  - Die zweiten 32-Bit beschreiben die Anzahl an Spalten-Matrizen.
  - Danach kommen die Zusammenhängenden Matrizen (zeilenweise). Die Zusammenhängenden Matrizen haben die Dimension des Beschleunigers
  - Vor jeder Zusammenhängenden Matrix ist ein 32-Bit Feld im Speicher frei, in dem die Adresse steht, wo die nächste Zusammenhängende Matrix anfangt minus Eins.

Folgende Funktionen wurden implementiert:

- Aufteilung einer zusammenhängenden Matrix in eine nicht zusammenhängende Matrix
- Multiplikation zweier Matrizen (zusammenhängend sowie nicht zusammenhängend)
- Maxpool von 3x3 und 5x5 zusammenhängenden Matrizen

2.5 Quellen

[1] David M. Harris and Sarah L. Harris, "Digital Design and Computer Architecture, RISC-V Edition"http://pages.hmc.edu/harris/class/e85/old/fall21/lect23.pdf
zuletzt aufgerufen am: 23.03.2022

[2] David A. Patterson, John L. Hennessy, "Computer Organization and Design RISC-V Edition: The Hardware Software Interface", Morgan Kaufmann, 2017.

[3] ISA Specification RISC-V https://riscv.org/technical/specifications/
zuletzt aufgerufen am: 03.01.2022

[4] https://github.com/TUD-CPU/PIPELINED-RISC-V
zuletzt aufgerufen am: 03.01.2022

[5] http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf
zuletzt aufgerufen am: 23.03.2022