Dokumentation zum Thema

FP6 Computation Unit

von Matthias Tappe, Timo Stapel, Niklas Thieme, Gero Grühn

Kurs:	Digital Design for Machine Learning
Unterthema:	Computation Unit
Betreuer:	Mikail Yayla
Bearbeitungszeitraum:	Wintersemster 2021/22

Inhaltsverzeichnis

1 Einleitung **2**

2 Addierer **3**
2.1 Halbaddierer 3
2.2 Volladdierer 3
2.3 Kogge-Stone-Addierer 3

3 Wallace-Tree-Multiplizierer **4**

4 Multiply-Accumulate-Unit **6**

5 Systolisches Array **7**

6 Asynchroner FIFO-Speicher **9**

7 Matrix-Multiplizierer **10**

8 Erweiterungen und Verbesserungen **11**
8.1 Pooling 11
8.2 Aktivierungs-Funktion 12
8.3 Controller 12

9 Fazit **13**

1 Einleitung

In dieser Dokumentation wird der Aufbau und die Entwicklung einer „Computation Unit“ für ein neuronales Netzwerk erläutert. Diese soll es ermöglichen effizient und schnell die erforderlichen mathematischen Operationen zu berechnen. Hierzu wurden die Aufgaben in kleinere Module aufgeteilt und bearbeitet. Im Verlauf des Projekts wurde ein Matrix-Multiplizierer für 8-Bit-Ganzzahlen (Zweierkomplement) in VHDL implementiert. Der Aufbau und die Funktionsweise werden im Folgenden erläutert.

2 Addierer

2.1 Halbaddierer

Der Halbaddierer addiert zwei einstellige Binärzahlen. Er hat zwei Ausgänge, Summe *(s)* und einen Übertrag *(c)*. Das Übertragssignal stellt einen Überlauf zur nächsten Stelle einer mehrstelligen Addition dar. Der Halbaddierer enthält ein XOR-Gatter für die Berechnung der Summe (s) und ein UND-Gatter für die Berechnung des Übertrags (c). Die boolesche Logik für die Summe (in diesem Fall s) ist a *XOR* b , während für den Übertrag (c) a *AND* b gilt.
Durch Hinzufügen eines ODER-Gatters zur Verknüpfung der Übertragsausgänge können zwei Halbaddierer zu einem Volladdierer kombiniert werden.

2.2 Volladdierer

Der Volladdierer addiert zwei Binärzahlen und einen möglichen Übertrag. Er hat drei Eingänge *(a, b, cin)* wobei a und b die Operanden sind und *cin* der Übertrag einer vorherigen Stelle. Die Schaltung erzeugt einen Zwei-Bit-Ausgang. Übertrag und Summe des Ausgangs werden durch die Signale *cout* und s dargestellt.

2.3 Kogge-Stone-Addierer

Der Kogge-Stone-Addierer ist ein Carry-Look-Ahead-Addierer (kurz: CLA-Addierer), mit zwei 8-Bit Eingängen *(a, b)* und einem 8-Bit Ausgang *(s)*. Das Diagramm (siehe Abbildung: 1) zeigt ein Beispiel für einen 4-Bit-KSA. Jede vertikale Stufe erzeugt ein Propagate- und ein Generate-Bit, wie dargestellt. Die Überträge werden in der letzten Stufe (vertikal) erzeugt, und dann mit den anfänglichen Propagierungsbits XOR-verknüpft, um die Summenbits zu erzeugen.

[1]Die Abbildungen zum KSA wurden übernommen von `https://en.wikipedia.org/wiki/Kogge%E2%80%93Stone_adder`, zuletzt aufgerufen am 21.03.2022

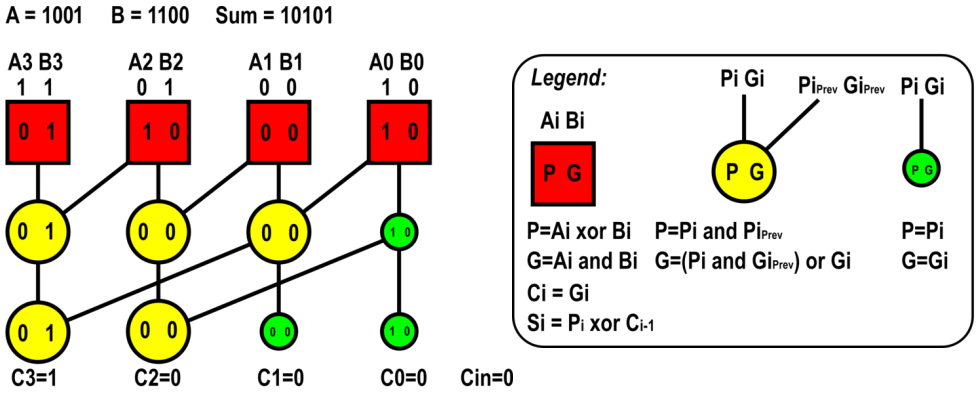

Abbildung 1: 4-Bit KSA[1] Beispiel und Legende

3 Wallace-Tree-Multiplizierer

Der Wallace-Tree-Multiplizierer ist eine Variante der langen Multiplikation. Der erste (1) Schritt besteht darin, jede Ziffer (jedes Bit) des einen Faktors mit jeder Ziffer des anderen zu multiplizieren. Jedes dieser Teilprodukte hat das gleiche Gewicht wie das Produkt seiner Faktoren. Das Endprodukt wird durch die gewichtete Summe all dieser Teilprodukte berechnet. Dieses Aufteilen ist simpel durch AND-Gatter realisierbar.
Das Endprodukt berechnet sich möglichst effizient, indem möglichst viele Bits möglichst parallel addiert werden. Hierfür werden Halb- und Volladdierer genutzt. Mit diesen lässt sich die Anzahl der Teilprodukte auf zwei reduzieren. Dieses Vorgehen wird im zweiten (2) Schritt angewand. Hierfür werden immer möglichst viele Bits (mit dem selben Stellenwert) in einen Volladdierer (oder Halbaddierer) eingeführtwas. Dadurch lassen sich drei zu zwei Zeilen reduzieren. Um Rechenzeit zu sparen schiebt sich der Übertrag nicht direkt hoch, sondern es entsteht eine neue Zeile für Überträge, die auch dementsprechend eine Stelle nach links rückt und in zukünftigen Additionen (mit den restlichen Zeilen) weiter reduziert wird. Dieses Vorgehen ist in Abbildung 2 dargestellt.
Der dritte (3) Schritt reduziert die zwei verbliebenen Zeilen zu dem Gesamtergebnis (also dem Produkt der Multiplikation). Hierfür wird ein komplexeres (im Vergleich zum Halb- und Volladdierer) Addierwerk gebraucht. In diesem fall nutzen wir den bereits beschriebenen Kogge-Stone-Addierer (KSA). Der dritte Schritt sieht vielleicht nicht so komplex aus, wie die vorherigen bei-

den Schritte, jedoch braucht dieser am längsten, da die Überträge direkt hoch rutschen und verrechnet werden müssen, was sehr zeitaufwändig, jedoch zwingend erforderlich ist, um das Ergebnis auf eine Zeile zu reduzieren, bzw. das Endergebnis (/Produkt) zu erhalten.

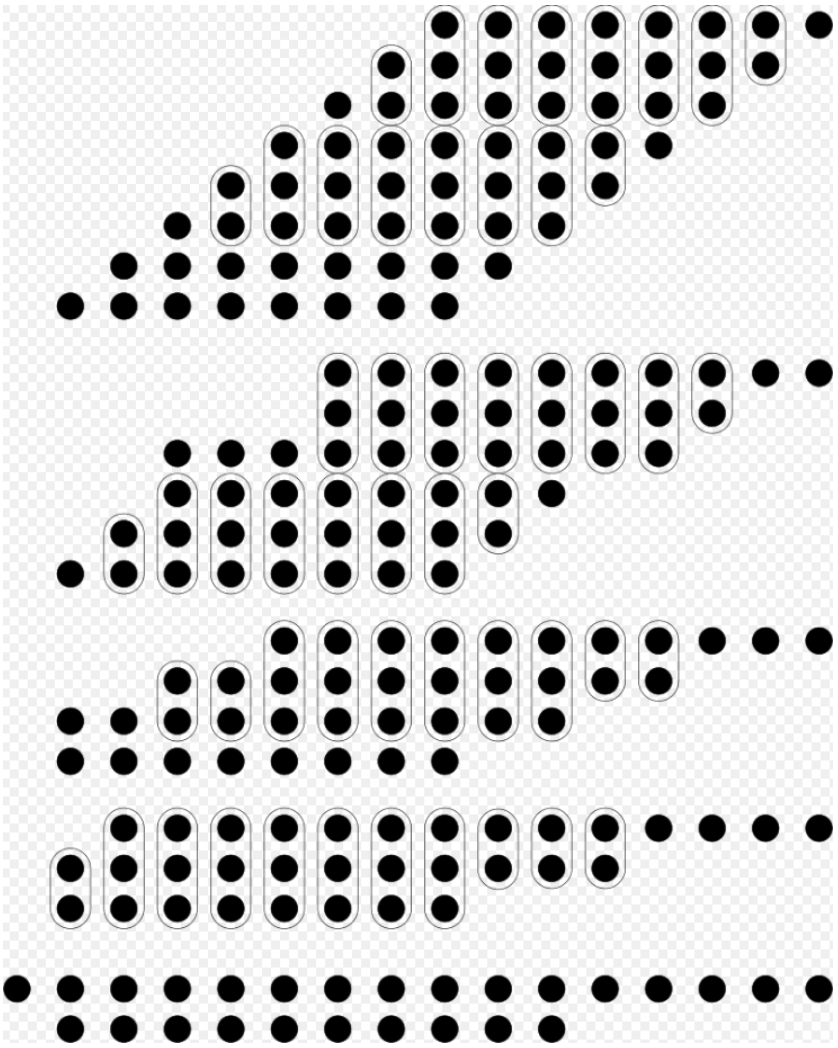

Abbildung 2: Wallace Tree[2] (8x8)

[2]Die Abbildung zum Wallace Tree wurde übernommen von `https://en.wikipedia.org/wiki/Wallace_tree`, zuletzt aufgerufen am 21.03.2022

4 Multiply-Accumulate-Unit

Die Multiply-Accumulate-Unit (kurz MAC-Unit), multipliziert die beiden 8-Bit-Eingangs-Zahlen, summiert diese Produkte und speichert dieses Ergebnis (Siehe Abbildung 3). Durch diese Operation kann eine Zelle einer Matrix-Multiplikation berechnet werden. Eine MAC-Unit kann mit unterschiedlichen Addierern und Multiplizierern realisiert werden, wobei wir uns für den Kogge-Stone-Addierer und den Wallace-Tree-Multiplizierer entschieden haben.
Der Wallace-Tree Multiplizierer ist, aus Performance-Gründen, eine sehr gute Wahl (kleines Design, kurze Verzögerung und geringer Energieverbrauch). Den Kogge-Stone-Addierer nutzen wir, da wir uns bereits gut mit dem Carry-Look-Ahead-Design (aus dem HaPra) auskennen, dieser einfach zu erweitern ist und CLA-Addierer effizient sind.
Des weiteren beinhaltet unsere MAC-Unit zwei weitere PIPO-Register (oder Speicher) um die beiden 8-bit-Eingangs-Zahlen mit einer Verzögerung von einem Takt, je vertikal oder horizontal, weitergeben zu können. Dies hilft uns bei der Erstellung eines Systolischen Arrays.

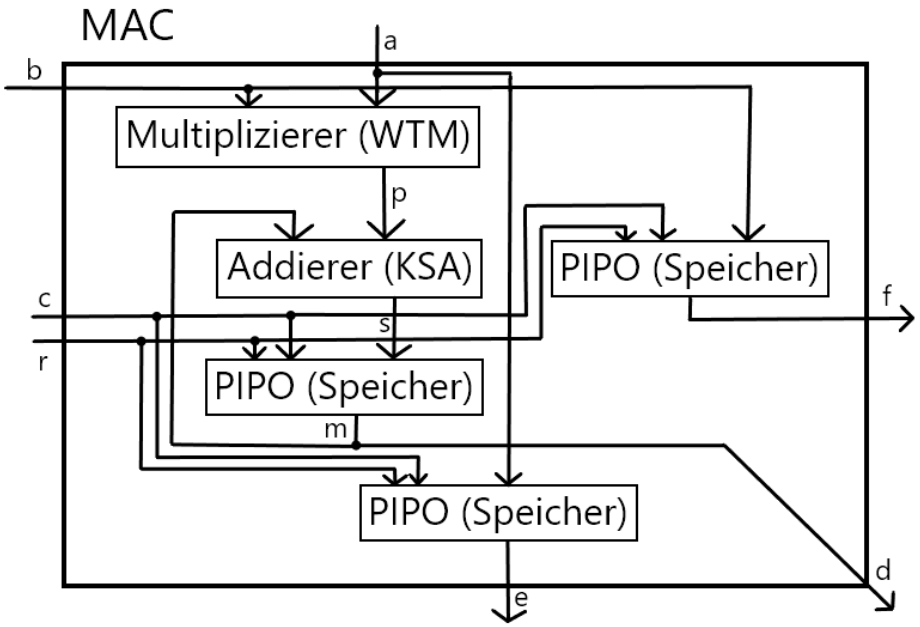

Abbildung 3: Schaltplan einer MAC-Unit

5 Systolisches Array

Das systolische Array ist ein matrixartiges Netz aus Rechenknoten welche in unserem Fall MAC-Units sind. Jeder Knoten gibt seine Daten nach der Bearbeitung (mit einem Takt Verzögerung) an seine Nachbarn weiter. So durchlaufen die Rechenwerte das systolische Array je vertikal (von oben nach unten) oder horizontal (von links nach rechts). Durch die Anordung der MAC-Units, welche in Abbildung 4 dargestellt ist, und das Durchfließen der Daten durch das Array, kann eine (aus Hardware-Sicht) effiziente Berechnung der Matrix-Multiplikation realisiert werden.
Hierbei ist zu beachten, dass jeder Eingangswert (*a(i)*, oder *b(j)*) eine 8-bit-Zweierkomplement-Zahl ist und mit jedem Takt neue Werte von links, bzw. oben nachrücken, um das systolische Array zu durchlaufen. Somit sind sowohl a, als auch b nicht nur Vektoren, sondern Matrizen. Die Anordnung wie die Werte das Array durchlaufen müssen ist, nicht beliebig, sondern strengstens festgelegt, damit sich in jeder MAC-Unit die Berechnung einer Zelle einer Matrixmultiplikation ergibt. Hierfür müssen je nach Zeile und größe der Matrix führende und/oder nachfolgende Null-Werte eingefügt werden. Somit werden die zu Multiplizierenden Matrizen in "Buffern" gespeichert und dem systolischen Array mit jedem Takt (Zeilen- bzw. Spaltenweise) zugeführt. Dies ist in Abbildung 5 dargestellt.
Unser systolisches Array ist dynamisch skalierbar und kann bei der Erstellung in einer beliebigen Größe erzeugt werden. Somit lässt sich dieses Rechengerät auf die maximale Größe der zu berechnenden Matrizen anpassen. Die Idee hinter der dynamischen Erstellung ist in Abbildung 4 zu sehen.

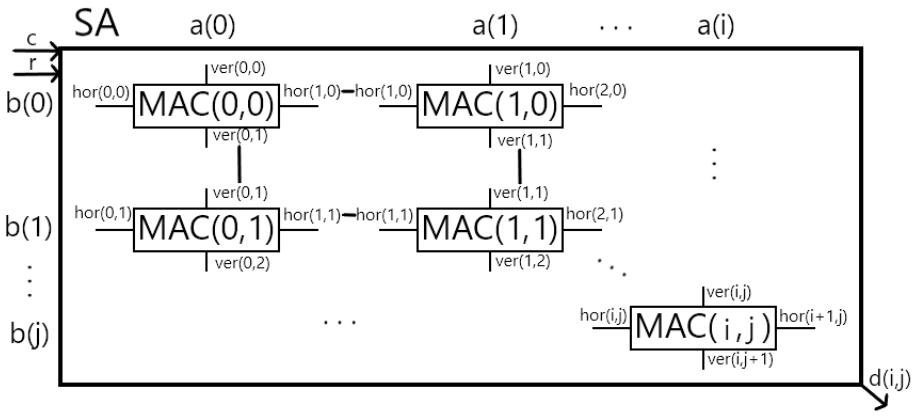

Abbildung 4: Schaltplan des Systolischen Arrays

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11} \cdot b_{11} + a_{12} \cdot b_{21} & a_{11} \cdot b_{12} + a_{12} \cdot b_{22} \\ a_{21} \cdot b_{11} + a_{22} \cdot b_{21} & a_{21} \cdot b_{12} + a_{22} \cdot b_{22} \end{pmatrix}$$

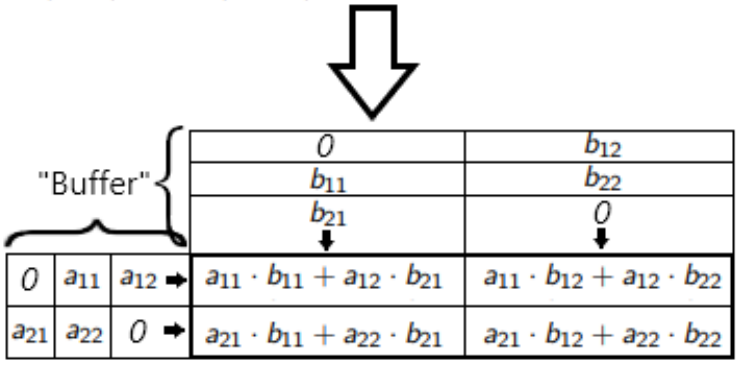

Abbildung 5: Beispiel Matrixmultiplikation und Befüllen der “Buffer“

6 Asynchroner FIFO-Speicher

Um die Buffer zu realisieren werden Stacks benutzt. diese werden mit Zeilen (links) oder Spalten (oben) der Matrizen gefüllt. Danach können die führenden Nullen hinzugefügt werden.
Dies ermöglicht vor allem ein möglichst einfaches Format der Dateneingabe von der CPU in den Matrix-Multiplizierer, da diese einfach zeilenweise (links) oder spaltenweise (oben) in den jeweiligen FIFO des Matrixmultiplizierers zu schreiben sind.
Um die Buffer zu befüllen und einen möglichst reibungslosen Ablauf zwischen der CPU und unserem Matrix-Multiplizierer zu gewährleisten, gibt es zum Schreiben und Auslesen der Matrizen FIFOs. Der FIFO hat zwei Clock-Domains und kann asynchron befüllt und gelesen werden. Es gibt Steuersignale um anzuzeigen, ob der Speicher voll oder leer ist, um Datenverluste oder fehlerhafte Daten zu verhindern. Eine konzeptuelle Abbildung des implementierten FIFO ist in Abbildung 6 zu sehen.

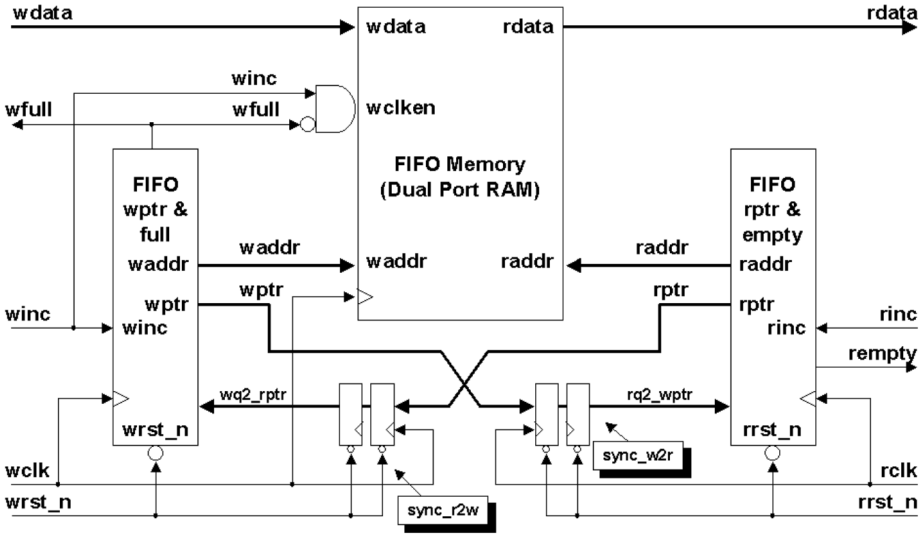

Abbildung 6: FIFO[3]

[3]Die Abbildung zum FIFO wurde übernommen aus `http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf`, zuletzt aufgerufen am 21.03.2022

7 Matrix-Multiplizierer

Der Matrix-Multiplizierer ist das komplette Konstrukt aus dem FIFO, den Stacks (oder Buffern) und dem Systolischen Array. Dieses wird durch den Controller gesteuert.
Um Die Matrizen aus den FIFOs korrekt (mit führenden Null-Werten) in die Stacks (oder Buffer) zu schreiben, haben wir den SA_Filler konstruiert.Um das Lesen des zweidimensionalen Ergebnisses aus dem systolischen Array, kümmert sich der SA_Reader. Dieser liest die Matrix Zeilenweise ein. Diese kann dadurch in einem eindimensionalen FIFO gespeichert und seriell abgerufen werden.
Der Controller kümmert sich darum, welche Aktion zur Zeit ausgeführt werden soll. Zur Kommunikation mit der CPU gibt es ein Steuerregister, aus dem der Controller, seine Steuerbefehle übernimmt.

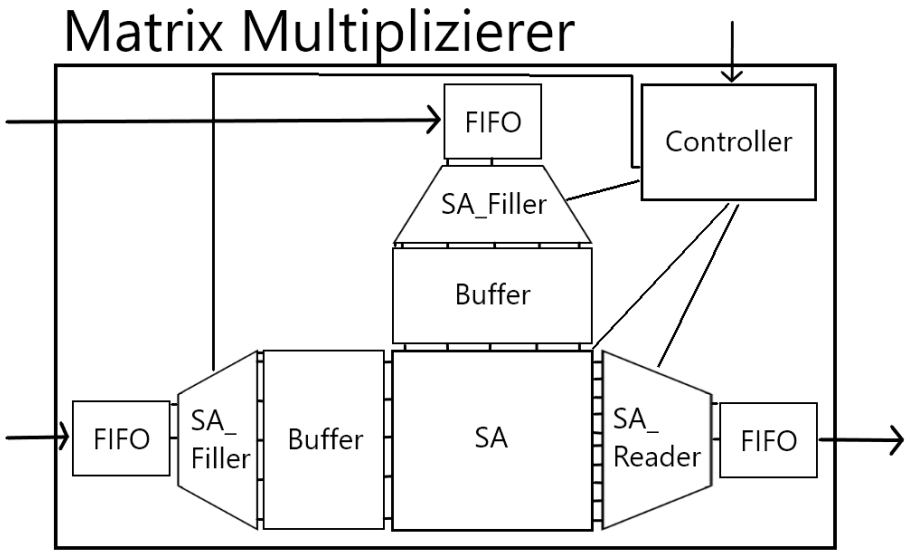

Abbildung 7: Matrix Multiplizierer

8 Erweiterungen und Verbesserungen

Man könnte den Matrix-Multiplizierer noch um Berechnungs-Einheiten für eine Pooling-Layer und eine Aktivierungs-Funktion erweitern. Somit würde man eine Hardware entwickeln, die alle Layer eines neuronalen Netzes berechnen kann. Ein Plan dafür ist in Abbilfung 8 dargestellt.

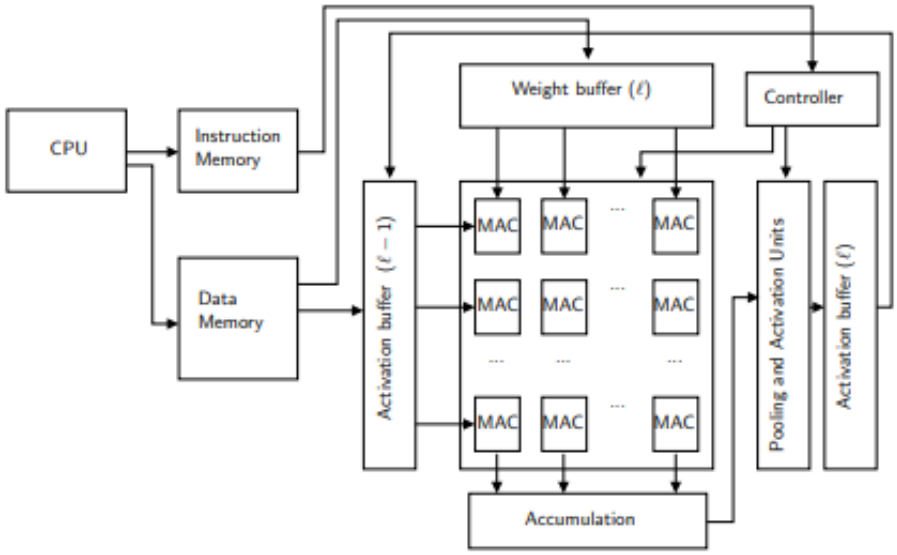

Abbildung 8: Hardware zur Berechnung von NNs[4]

8.1 Pooling

Beim Pooling geht es darum die Matrizen-Größe zu reduzieren und folgende Berechnungen zu vereinfachen. Dafür wird die Anzahl der Zellen einer Matrix reduziert, indem mehrere (benachbarte) Zellen zu einer Zelle zusammengefasst werden. So kann man beispielsweise mit einem 2x2-Filter beide Dimensionen einer Matrix halbieren, indem per Dimension zwei benachbarte Zellen zusammengefasst werden. Somit ergibt sich der 2x2 Filte und je vier Zellen werden zu einer zusammengefasst.

[4]Die Abbildung der Hardware zur Berechnung von NNs wurde übernommen aus einer Präsentation zu diesem Fachprojekt von Mikail Yayla

Die Art des Zusammenfassens kann theoretisch ausgesucht werden. Wir würden uns auf das (etablierte) Max-Pooling festlegen. Dabei wir der hochste Wert aus dem Filter-Bereich übernommen (und die Anderen verworfen).
Somit bäuchte man Hardware die iterativ eine Matrix durchläuft und einen Filter auflegt, bei dem die Zellen-Werte innerhalb des Filters miteinander Verglichen werden (→ Vergleicher) und den größten Wert ausgibt und aus diesen Ergebnis-Werten eine neue, kleinere Matrix erstellt. (→ 2D-Adressierbarerer Speicher)
Wenn man sich auf ein spezielles Pooling-Verfahren festlegt, ist dies sicherlich auch auf einer skalierbaren Matrix-Größe umsetzbar. Und könnte mit unserem Matrix-Multiplizierer zusammen funktionieren.

8.2 Aktivierungs-Funktion

Um die Aktivierungs-Funktion eines Neurons zu berechnen, müssen alle eingehenden Kanten eines Neurons aufsummiert werden. Dies entspricht den Spalten in der Ergebnismatrix, wenn die Gewichtsmatrix von oben eingegeben wurde. Diese Summe über- oder unterschreitet dann einen gewissen Schwellenwert, der in der Aktivierungs-Funktion gegeben ist. Wird dieser Wert überschritten, gilt das Neuron laut der Funktion als “aktiv“, sonst als “inaktiv“. Dafür weist die Funtkion diesem Neuron bestimmte (feste) Ausgangswerte zu, welche dann in die Berechnung der nächsten Ebenen einfließen.
Somit bräcuhte man Hardware zur Summierung von Spalten einer Matrix (→ Addierer) und Hardware zum Vergleichen der Summe mit dem gegebenen Schwellenwert (→ Vergleicher).
Wenn man sich auf eine spezielle Aktivierungs-Funktion festlegt, ist dies sicherlich auch auf einer skalierbaren Matrix-Größe umsetzbar. Und könnte mit unserem Matrix-Multiplizierer zusammen funktionieren.

8.3 Controller

Des weiteren bräcuhte man einen Controller, der dieses Gesamtkonstrukt steuert. Hierfür muss dieser die benötigte Rechenoperation ausführen können, indem er die benötigte Rechenkomponente ansteuert.
Zudem sollte der Controller die Größe der berechneten Matrix kennen, da durchaus auch Matrizen berechnet werden können, die kleiner als die maximale Größe sind.

Außerdem ist die kommunikation des Controllers mit der CPU essenziell, damit diese die benötigten Daten (Matrizen) für die Berechnungen, zur korrekten Zeit in den richtigen FIFO schreiben kann.

9 Fazit

Im Verlauf des Projekts wurde ein „Matrix-Multiplizierer“ im kleinen Maßstab in „VHDL“ implementiert. Die Module wurden zunächst getrennt voneinander entwickelt und anschließend zu einem ganzen zusammengesetzt.
Durch die Implementierung in kleinen Modulen, war es uns möglich die Aufgaben zu verteilen und effizient zu bearbeiten. Wobei dabei, gerade gegen Ende (wo es komplexer wurde), immer mehr Probleme aufgetreten sind. Beispielsweise hat sich das zusammenfügen komplexerer Komponenten, die für sich funktioniert haben. Bei denen es dann beim Zusammenfügen, vermehrten zu kleinen Problemen kam, als wirklich zeitraubend ergeben.
Gerade auch die Skalierbarkeit der Größe des Matrix-Multiplizierers hat die Umsetzung erheblich erschwert, da Packete mit speziellen Typen für Signale/Verbindungen erstellt werden mussten. Mit diesen Themen kannte sich vorher keiner von uns aus und jeder musste ich erst einmal viel Zeit nehmen um sich in diesen Bereich von VHDL hinein zu arbeiten.
Abschließend können wir sagen, dass wir eine ganze Menge über neuronale Netze und deren Hardwaretechnische berechnung gelernt haben. Und wir generell sehr viel Spaß am Hardware-Design gefunden haben.
Generell Ideen zu entwickel um bestimmte Probleme, wie beispielsweise das Befüllen der Buffer vor dem systolischen Array, zu beheben waren anspruchsvoll, mit viel Zeitaufwand verbunden, aber wir haben uns stets gefreut wenn wir voran gekommen sind.
Abschließend betrachtet war dies also ein schönes Projekt für unsere Gruppe.