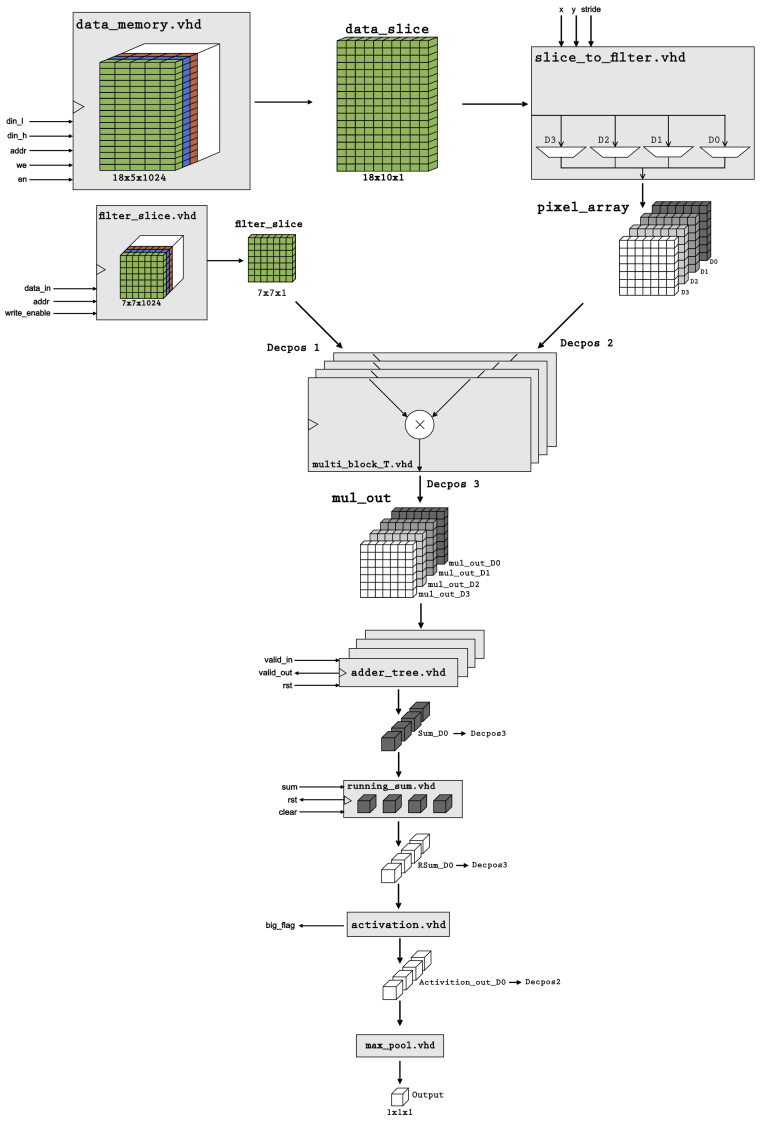
CHALMERS



FPGA-implementation av ett neuralt nätverk

Examensarbete inom data- och informationsteknik.

TIM JADEGLANS
HANNA LARSSON

Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2019

Bachelor thesis

FPGA implementation of a neural network

TIM JADEGLANS
HANNA LARSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University Of Technology
University of gothenburg
Gothenburg, Sweden 2019

FPGA-implementation av ett neuralt nätverk

Handledare:
August Von Hacht, Synective Labs
Joel Klingspor, Synective Labs
Arne Linde, Chalmers tekniska högskola
Examinator:
Peter Lundin, Institutionen för data- och informationsteknik

Examensarbete 2019
Institutionen för data- och informationsteknik
Chalmers tekniska högskola / Göteborgs Universitet
SE-412 96 Göteborg
Telefon +46 31 772 1000

Omslagsbild: Grafisk representation av funktionsblocken som utgör den implementerade arkitekturen.

FPGA implementation of a neural network

Tim Jadeglans, Hanna Larsson
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Image recognition is a quickly growing field where convolutional neural networks, CNN, are in the bleeding edge. Today fast GPUs are used which consume a lot of power. Field programmable gate arrays, FPGAs, are more energy efficient per calculation. This report describes an architecture of a convolutional neural network implemented in a field programmable gate array. The main purpose is to design an architecture and demonstrate its functionality in regards to power, speed and resource usage. In order to achieve the architecture, the project has followed general guidelines for a convolutional neural network, with filters that extend over the entire depth of the image.

The parameters of the design were adapted for the FPGA used in the project. The dimensions of the memory were adjusted to reduce the number of times each data has to be loaded for each calculation, due to max-pooling. The final architecture, however, resulted in a flexible enough design that is adaptable to other FPGAs. When implemented, the calculations used both data and filters from a limited read-only memory, ROM, the design could use data from the main processor.

The computing capacity of the architecture is far below the theoretical capacity of the FPGA. However, there are multiple possibilities for improvements which would improve the computing potential dramatically. To utilize the increased potential, the summing tree used in the architecture can be modified which will potentially double the calculations per clock cycle and optimize the critical data path to further increase the clock speed. Despite these limitations, the current architecture has higher performance-to-power ratio than a GTX 1060.

Keywords: Convolutional Neural Network, CNN, Field Programmable Gate Array, FPGA, Image Recognition.

Sammanfattning

Bildigenkänning är ett snabbt växande område där så kallade *convolutional neural networks*, CNN, ligger i framkant. Idag används främst GPU:er som är snabba men använder mycket energi. FPGA:er, *field programmable gate array*, är mer energieffektiva. Denna rapport beskriver en arkitektur av ett CNN tillämpat i en FPGA. Syftet är att skapa en arkitektur och påvisa dess funktionalitet i avseende på strömförbrukning, beräkningskapacitet och resursanvändning. Arkitekturen karakteriseras av en generell CNN-arkitektur med filter som sträcker sig över hela djupet av bilden.

Arkitekturens parametrar är anpassade för den FPGA som arbetats mot. Bland annat är huvudminnets storlek anpassad för att minimera mängden data som behöver laddas in, samt parallella beräkningar för att effektivt göra max-pooling. Detta har resulterat i en arkitektur som är flexibel nog för att kunna anpassas till andra FPGA:er. Beräkningsenheten använder sig av data och filter från ett begränsat *Read only memory*, ROM, men kan även använda sig av data från en huvudprocessor.

Beräkningskapaciteten ligger långt under den teoretiska maxkapacitet FPGA:n har, det finns dock flera förbättringspunkter som skulle öka potentialen mycket. För att kunna utnyttja den ökade potentialen kan framtida arbeten använda sig av ett bättre summeringsträd som skulle dubbla beräkningarna per klockcykel samt optimera den kritiska datavägen för att öka klockhastigheten. Trots begränsningarna är den nuvarande arkitekturen ändå mer energieffektiv än den motsvarande grafikprocessorn GTX 1060 per beräkning.

Förord

Examensarbetet utfördes under vårterminen 2019 av Tim Jadeglans och Hanna Larsson på Elektroingenjörsprogrammet 180hp på Chalmers tekniska högskola. Arbetet omfattar 15hp och skrevs på institutionen för data- och informationsteknink.

Inledningsvis vill vi tacka de som hjälpt oss med examensarbetet. Arne Linde, handledare från institutionen för data- och informationsteknink, som bidragit med bra åsikter och feedback gällande arkitekturen. Ett extra tack till August Von Hacht och Joel Klingspor, från Synective Labs, som stöttat och hjälpt till genom hela arbetsprocessen.

Tim Jadeglans & Hanna Larsson, Göteborg, maj 2019

Innehållsförteckning

Beteckningar

bRAM - Block Random-Access Memory
Minnesblock där alla adresserna går att läsa och skriva till utan att påverka andra adresser.

CNN - Convolutional Neural Network
En typ av neuralt nätverk som ofta används för bildbehandling.

DSP - Digital Signal Processor
Enhet för signalbehandling. Används för att utföra multiplikationer.

FPGA - Field Programmable Gate Array
En programerbar krets som kan utföra logiska operationer genom att ändra dess fysiska funktion.

FLOPS - Floationg Point Operations per second
Ett mått på beräkningskapacitet för flyttal.

MACS - Multiply-Accumulate per second
Ett mått på beräkningskapacitet för heltal.

GPU - Graphics Processing Unit
Grafikprocessor som även kan göra parallella beräkningar.

LUT - Lookup-table
En lista med förutbestämda svar för alla indata.

ReLU - Rectified Linear Units
funktionen max(0,x) som används som en snabb aktiveringsfunktion.

ROM - Read only memory
Ett minne som enbart går att läsa från.

S1 & S2 - Stride 1 och Stride 2
Metod för att minska beräkningar och utdata.

1
Inledning

Bildigenkänning har ett flertal användningsområden. I ett tidigare arbete har bildigenkänning använts som lösning för att övervaka brottslingar och identifiera vapen från luften med hjälp av drönare [1]. Då bildigenkänning är ett problem som är svårt att åstadkomma med traditionell mjukvara, kan neurala nätverk vara ett effektivare sätt att lösa problemet. Ett neuralt nätverk fungerar som en komplex funktion som istället för att använda en känd modell, tränas med hjälp av exempeldata. Stora mängder data samlas från fältet eller experiment som används för att träna nätverket. Träningen sker generellt inte på fält utan i stora datorcenter. Ett *convolutional neural network*, CNN, är ett neuralt nätverk med tränade filter som känner igen drag så som kanter och hål. Styrkan kommer dels i att dragen som nätverket letar efter inte är programmerade, utan inlärda från träningen, och när fler lager används så kan nätverket hitta mer komplexa drag såsom ögon.

Rapporten är skriven på uppdrag av Synective Labs som specialiserar sig på signal- och bildbehandling, hårdvaruacceleration och inbyggda system baserade på FPGA-tekniken och GPU-användning [2]. En *field programmable gate array*, FPGA, kan användas till att accelerera ett system genom att avlasta ett neuralt nätverk från en huvudprocessor till en FPGA. Traditionellt har grafikprocessorer, GPU:er, använts för att accelerera neurala nätverk men då de är energikrävande ser Synective Labs FPGA-tekniken som ett alternativ, för drönare och andra enheter som har en begränsad energitillgång, då lågeffekts-GPU:er inte finns på marknaden.

1.1 Syfte och mål

Projektets syfte är att skapa en arkitektur som fungerar som ett CNN och påvisa dess funktionalitet avseende energiförbrukning, beräkningskapacitet och resursanvändning.

1.2 Avgränsningar

I projektet kommer inte nätverket tränas. Enbart tillfälliga vikter används för att kontrollera att resultatet är korrekt. Systemet kommer inte att ta in data utifrån utan bara köra på vikter från ett filter och en delmängd av en hel bild.

1.3 Förutsättningar

Förutsättningar till projektet bestämdes av hårdvaran som arbetades mot och mjukvaran som tillämpades. I detta avsnitt beskrivs dessa delar mer ingående.

1.3.1 Hårdvara

Den hårdvara som användes för projektet är ett utvecklingskort från Xilinx, *ZedBoard Zynq-7020*. De resurser som är tillgängliga på kortet är en Artix-7 FPGA med 220 stycken DSP-slices, 140 stycken 32-bitars eller 280 stycken 16-bitars bRAM, 53.2k LUTs samt en dual-core ARM Cortex-processor. Det finns även stöd för externt minne. Mängden DSP-slices sätter en över gräns på mängden beräkningar som kan utföras per klockcykel och mängden bRAM begränsar hur många filter och hur stor del av en bild som kan lagras i FPGA:n i taget.

1.3.2 Mjukvara

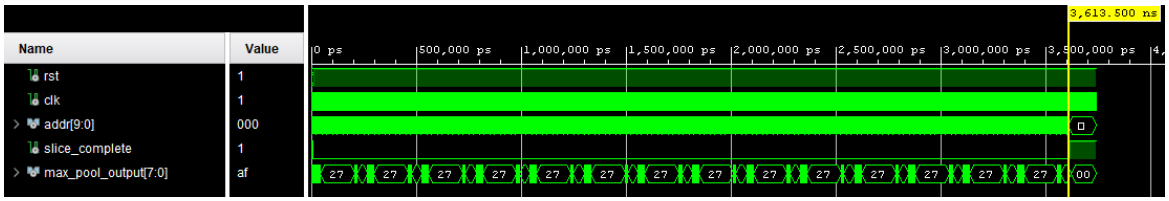
Vivado Design Suite [3] är den programvara som användes för att skriva programkod i det hårdvarubeskrivande programspråket VHDL, simulera och syntetisera designen. För att säkerställa att programkoden gör det som tänkt, testades funktionaliteten med hjälp av simulering. När programkoden klarat funktionstesterna syntetiserades koden. Syntetisering innebär att designen översätts till motsvarande grindrepresentation, där det i loggar visar hur mycket resurser blocket kräver. Eftersom utvecklingskortet har en begränsad mängd hårdvaruresurser för ändamålet, är det viktigt att storleken på blocken är valda så att den fullständiga arkitekturen går att köra. Denna process, med simulering och syntetisering, slutfördes för varje funktionsblock, innan de kopplades ihop. Detta för att underlätta felsökning och för att säkerställa funktionen för hela arkitekturen.

1.4 Metod

Arkitekturen är baserad på den generella uppbyggnaden av ett *Convolutional Neural Network*. Designprocessen inleddes med att projektet delades in i mindre funktionsblock. Varje funktionsblock funktionstestades separat för att sedan kopplas ihop till en fullskalig arkitektur. Den teoretiska prestandan räknades fram för att sedan jämföras med verkliga resultat samt förbättringsmöjligheter.

1.4.1 Mätningar

För att mäta prestandan användes Vivados verktyg för simulering och syntetisering med ett *read only memory*, ROM. Detta implementerades istället för att läsa informationen från en extern källa. Beräkningshastigheten beräknades genom att simulera en testslice med ett testfilter och räkna antalet klockcykler tills flaggan *slice complete* sätts hög för att indikera att alla beräkningar är färdiga. Ett exempel på simuleringsresultatet visas nedan i figur 1.1.



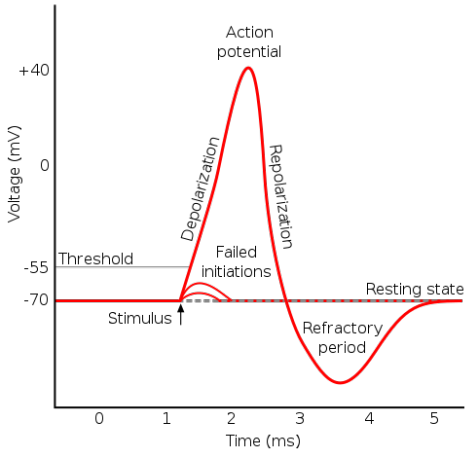
Figur 1.1: Exempel på simuleringsresultat från arkitekturen där *slice complete* flaggan sätts hög vid tiden 3613,5 ns.

2

Neurala nätverk

I detta avsnittet kommer de teoretiska delarna för arkitekturen beskrivas övergripande. Dessa delarna är karakteristiska för ett *Convolutional Neural Network*. Ett artificiellt neuralt nätverk efterliknar uppbyggnaden av det biologiska nervsystemet. En biologisk nervcell avger en liten elektrisk signal vid stimuli[4]. Denna signal varierar i både frekvens och amplitud. Signalens karakteristik utgörs av fyra olika faser; stimuli, depolarisering-, repolarisering- och hyperpolariseringsfasen.

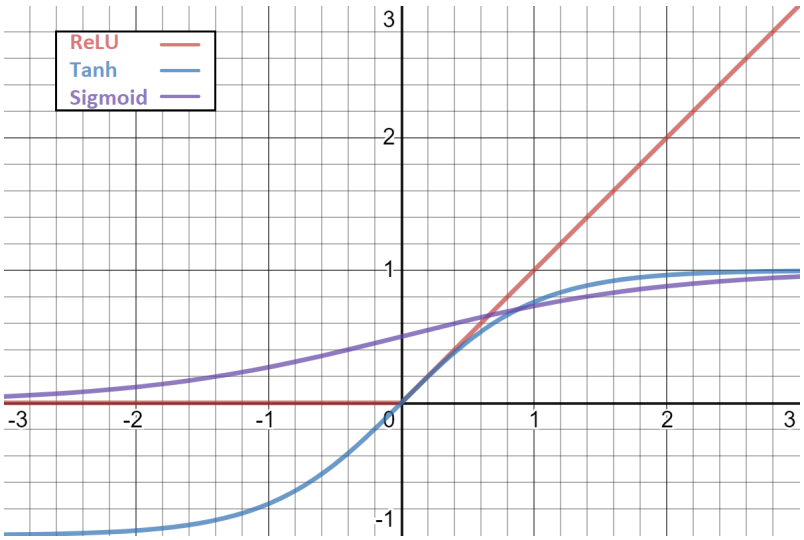
Den elektriska signalen som den biologiska nervcellen utgör beror på de olika jonkoncentrationerna innanför och utanför cellmembranet. Denna koncentrationsskillad skapar en vilopotential. När stimuli är tillräckligt stor, större än tröskelpotentialen, inleds depolariseringsfasen då potentialen stiger. När potentialen är hög nog ger nervcellen ut en elektrisk impuls vilket påverkar andra nervceller. Signalens karakteristik illustreras i figur 2.1. Tidigare i avsnittet nämndes att ett CNN efterliknar det biologiska nervsystemet och några av huvuddelarna kommer att beskrivas i avsnitt 2.1-2.3.



Figur 2.1: De fyra karakteristiska faserna för en biologisk nervcell vid stimuli. Tröskelpotential vid -55mV [5].

2.1 Aktiveringsfunktion

Aktiveringsfunktionens syfte är att efterlikna tröskelpotentialen hos en nervcell. Detta innebär att nätverket kommer utesluta information som inte uppfyller rätt värdemängd. Det finns många olika typer av aktiveringsfunktioner, såsom Sigmoid, Tanh och ReLU. Den sistnämnda användes i projektet och är vanligt förekommande[6]. En av anledningarna till detta är främst för dess simplicitet samtidigt som den påvisar bra resultat. ReLU-funktionen beskrivs nedan, ekvation 2.1 och visualiseras även i figur 2.2.



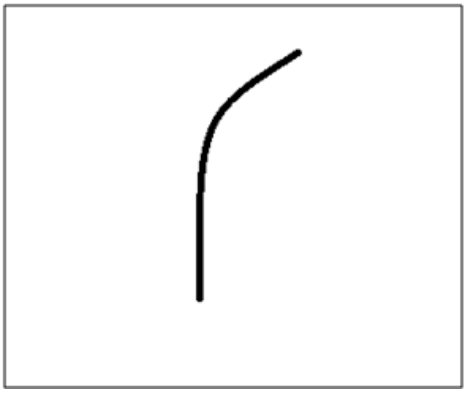
Figur 2.2: ReLU-funktionen, f(x)=max(0,x) där f(x)=0 då x<0, jämförd med Tanh- och Sigmoid-funktionen, plottad i intervallet -3 till 3.

$$f(x) = \begin{cases} 0 & \text{för } x < 0 \\ x & \text{för } x \geq 0 \end{cases} \tag{2.1}$$

2.2 Filter

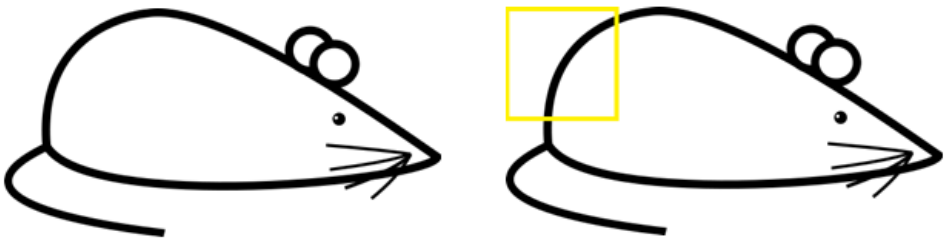
För att kunna avgöra vad bilden innehåller används filter. Dessa filter är ett resultat av en stor mängd träningsdata som består av generella drag av objektet som skall identifieras. I detta avsnittet presenteras ett nedskalat exempel över filterfunktionen. Till vänster i figur 2.3, visas ett exempel på ett filter för att hitta en kant, i exemplet används ett filter med dimensionerna 7x7x1. Där den tredje dimensionen motsvarar djupet, vilker i detta fall är 1 då bilden är svartvit, men skulle också kunna vara 3 med ett rött, grönt respektive blått lager, RGB. Till höger är filtrets motsvarighet utan de faktiska pixelvärdena.

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0



Figur 2.3: Ett filter med dimensionerna 7x7x1 och dess grafiska motsvarighet utan pixelvärden makrerade [7].

Filtret kommer att multipliceras för alla segment i bilden för att objektet skall kunna identifieras, i detta exempel visas den inlästa bilden i figur 2.4. Där det finns likheter, det vill säga där kanten passar in, kommer ett stort värde returneras, detta området är gulmarkerat, jämför figur 2.3 och 2.4.



Figur 2.4: Den inlästa bilden som skall identifieras. Sektionen markerat med gult är arean där filtret har mest likheter[7].

I figur 2.5 illustreras multiplikationen för det markerade området. Till vänster är delen av orginalbilden samt dess pixelvärdesrepresentation och till höger filtret som bilden skall multipliceras med. Med följande beräkningar kan det totala värdet från segmentet beräknas genom att summera värdet från alla multiplikationer:

$$(50 \cdot 30) + (50 \cdot 30) + (50 \cdot 30) + (20 \cdot 30) + (50 \cdot 30) = 6600$$

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Figur 2.5: Till vänster är det inlästa segmentet av bilden vilken skall multipliceras med filtret. I mitten, det inlästa segmentets pixelvärdesrepresentation och till höger filtret. Finns det många likheter mellan bilden och filtret returneras ett högt resultat efter multiplikationen och summeringen[7].

I nästa del av exemplet multipliceras filtret med en annan del av bilden. Denna gången finns det inga likheter alls mellan filtret och bilden, vilket innebär att multiplikationen och summeringen, i detta exemplet, kommer att returnera värdet 0. Denna principen gör det möjligt att identifiera objekt i bilden. Antalet filter som körs varierar beroende på arkitektur och prestanda som skall uppnås.

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

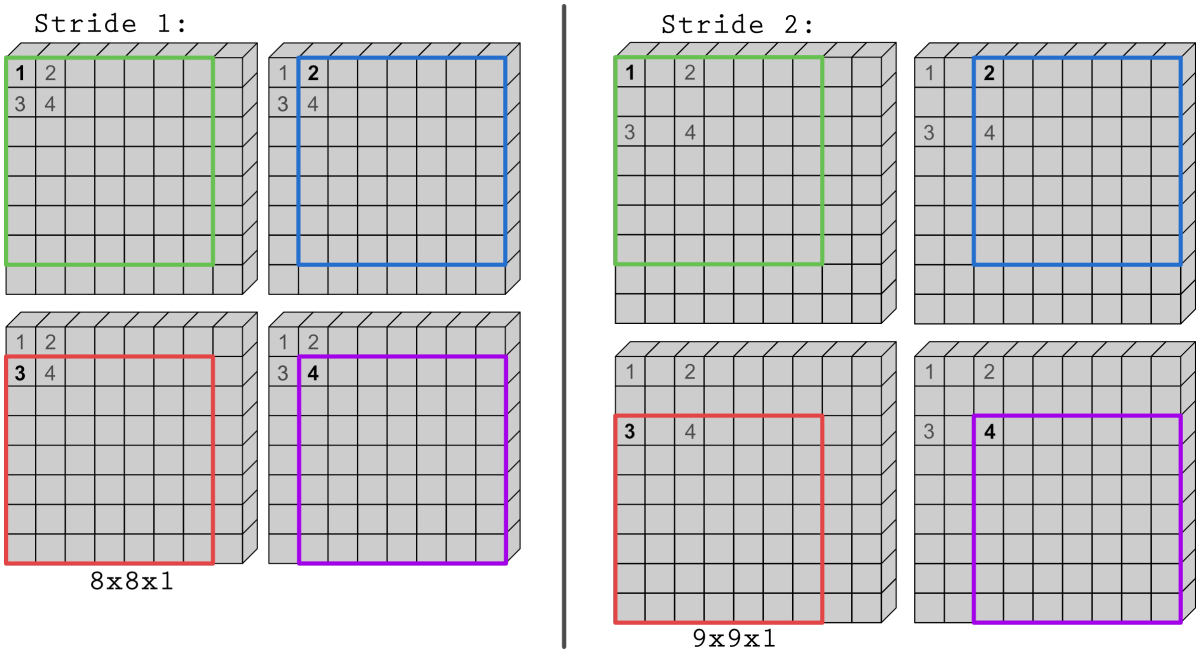
*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Figur 2.6: Till vänster är det inlästa segmentet av bilden som skall multipliceras med filtret. I mitten, det inlästa segmentets pixelvärdesrepresentation och till höger filtret. Finns inga eller några likheter mellan bilden och filtret returneras ett lågt resultat efter multiplikationen och summeringen[7].

2.3 Stride

För att minska beräkningsmängden och därmed storleken på utdatan, implementeras en stridefunktion. Stride 1 (S1) innebär att filtret kommer att appliceras på varje inläst pixel. För att minska antalet beräkningar används stride 2 (S2), där filtret appliceras på vartannan inläst pixel, se figur 2.7. Stride 3 och stride 4 fungerar på samma sätt, vilket innebär att filtret appliceras på vart tredje respektive fjärde pixel. Högre stride används oftast för lagren med stor yta. Att använda för hög stride kan leda till att beräkningsresultatet för hela bilden försämras. Därför anpassas strideantalet efter arkitekturen. I detta arbetet kommer S1 och S2 kunna köras.

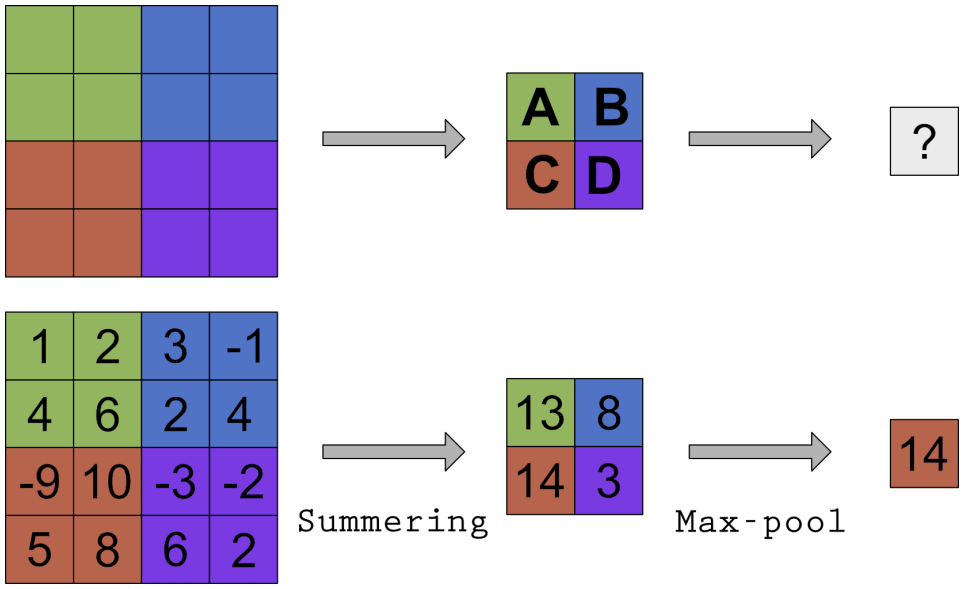


Figur 2.7: Stride 1 respektive stride 2. Där siffrorna 1-4 representerar koordinaterna för varje hörn där ett filter med dimensionerna 7x7 applicerats. För att tydliggöra varje filterapplicering är detta markerat med grönt, blått, rött samt lila.

2.4 Max-pool

Max-pooling implementeras för att minimera datamängden i arkitekturen. Det är ett eget filter som appliceras på hela bilden. I denna arkitekturen finns möjligheten att använda 2x2 max-poolingfilter. För att möjliggöra detta måste convolution-filtret appliceras på fyra områden. I figur 2.8 appliceras ett 2x2 convolution-filter med stride 2 på 4 områden. Stride förklaras i kapitel 2.3. Efter att filtret applicerats fyra gånger, summeras samtliga pixelvärden i vadera filtersektion. För att tydliggöra konceptet visas beräkningarna nedan för varje segment:

$$A = 1 + 2 + 4 + 6 = 13$$
$$B = 3 - 1 + 2 + 4 = 8$$
$$C = -9 + 10 + 5 + 8 = 14$$
$$D = -3 - 2 + 6 + 2 = 3$$

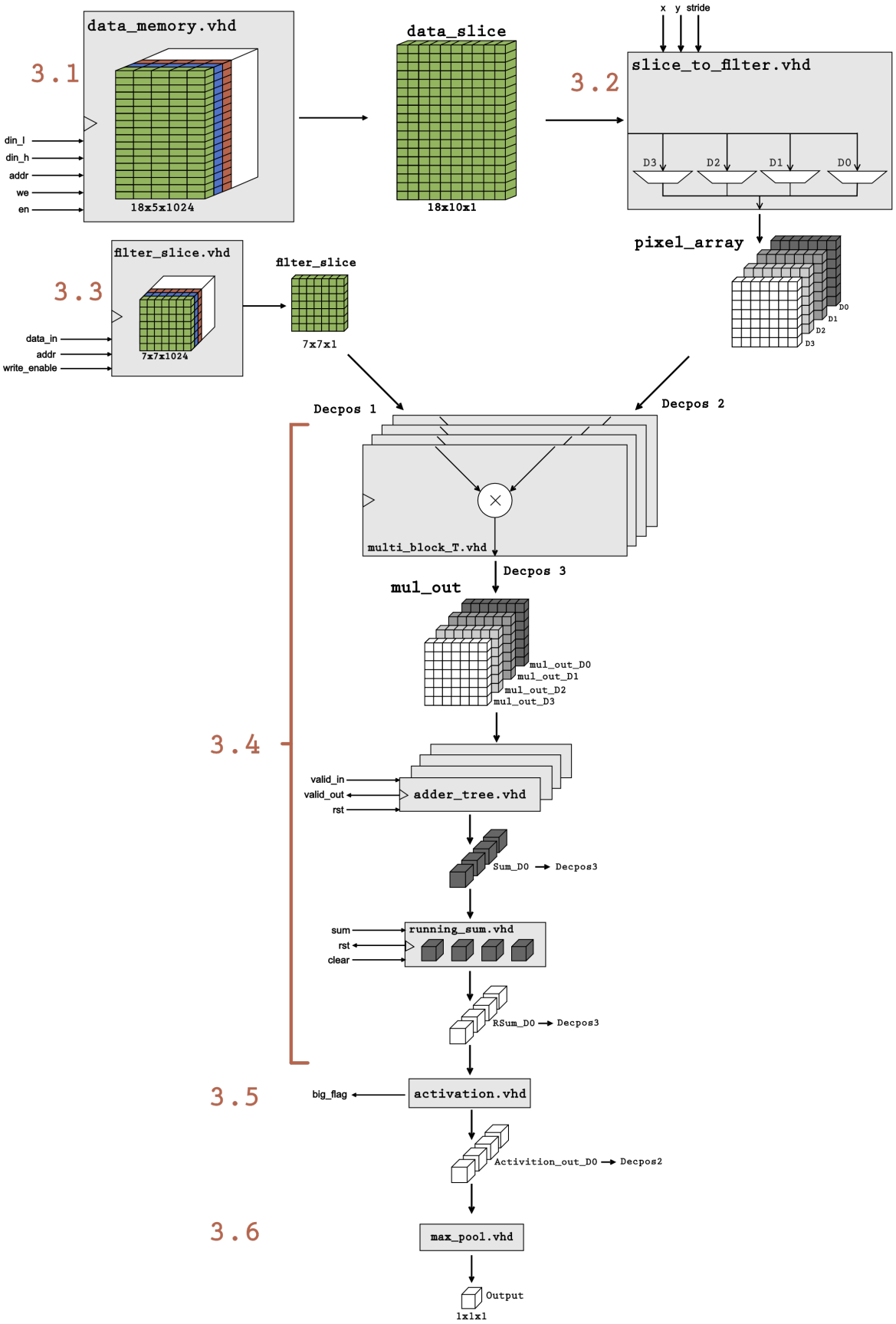


Figur 2.8: Summering av alla elementen i varje filtersektion där det största resultatet returneras med hjälp av max-pooling.

Efter summeringen returneras det största värdet och ett nytt område läses in. I arkitekturen som implementerats i projektet är dimensionerna på det inlästa blocket 18x10, storleken på convolution-filtret 7x7 och storleken på max-poolfiltret är 2x2.

3
Arkitektur

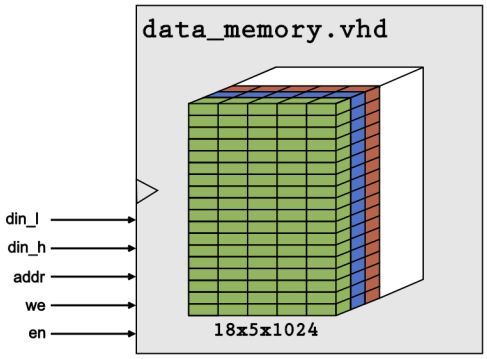
Arkitekturen i figur 3.1 är uppbyggd sådant att vikterna och datat multipliceras elementvis. I minsta form krävs en plats att lagra data, lagra filter, multiplicera dessa värden samt summerar resultatet. För att göra det mer effektivt är dataområdet större än filterområdet vilket gör att *Slice_to_filter.vhd* i figur 3.1 behövs för att välja vad som ska gå vidare till filtermultiplikationen. *Slice_to_filter.vhd* och *Data_memory.vhd* gör det möjligt att flytta på filtret och är beskrivet i kapitel 3.1 och 3.2. Multiplikationen sker i *multi_block_T.vhd* och skickar ut 49x4 stycken tal som motsvarar filtret elementvis multiplicerat med bilddatat. Dessa 49 tal summeras i *adder_tree.vhd*. För att kunna summera fler än de 49 tal vilket är i en dataslice, lagras resultatet i *running_sum.vhd* som summerar ihop alla resultat för filtret. Multiplikationen och summeringen utgör beräkningsblocket vilket beskrivs i kapitel 3.4. När beräkningarna för hela filter- och datadjupet är utförda finns fyra resultat efter aktiveringsfunktionen. Efter max-poolingfunktionen returneras ett resultat om det efterkommande lagret är ett max-poolinglager.



Figur 3.1: Den implementerade arkitekturen. Den röda numreringen symboliserar vilket textavsnitt som funktionsblocket tillhör.

3.1 Data_memory

Syftet med funktionsblocket *data_memory.vhd* är att kontinuerligt läsa in data från en bild. Inläsningen till funktionsblocket sker med hastigheten 32 bitar per klockcykel, vilket är den största begränsande hastighetsfaktorn för hela arkitekturen. Eftersom hela bilden inte får plats i FPGA:n, sker inläsningen i segment. För att avlasta databussen optimeras inläsningen för varje minnesplats, vilket medför att inläsningen sker minimalt antal gånger. För att åstadkomma detta storleksoptimeras blocket. De faktorer som begränsar dimensionerna för *data_memory.vhd* är filterstorleken (7x7), stride, max-pooling och antalet *block random access memory*, bRAM, som finns tillgängligt. För hela arkitekturen finns 280 stycken 16-bitars bRAM, varav 49 används för lagring av filter. Funktionsblocket visualiseras nedan i figur 3.2.



Figur 3.2: Funktionsblocket *data_memory.vhd*.

3.1.1 Dimensionsberäkningar S1

Designen och beräkningsprocessen utgick från blockets minsta möjliga dimensioner. Då arkitekturen är anpassad efter att köra stride 2, måste även *data_memory.vhd* vara det, eftersom stride 2 kräver fler inlästa minnesplatser än stride 1. Arkitekturen är även designad för att senare i processen göra max-pooling, det innebär att filtret måste appliceras fyra gånger på det inlästa området. För att möjliggöra inläsningen med stride 2, måste blockets minsta dimensioner vara 9x9, se figur 2.7 i avsnitt 2.1.4. Värdet som läses in är ett 8-bitars tal och eftersom varje bRAM rymmer 16 bitar kan två tal lagras på varje minnesplats. Till följd av detta bestämdes blockets bredd till 5, vilket resulterar i den slutgiltiga bredden 10 som uppfyller kraven för stride 2. Varje minnesblock rymmer 1024 adresser vilket ger maxdjupet 1024. Maxhöjden, h_{max}, blocket kan anta blir därför:

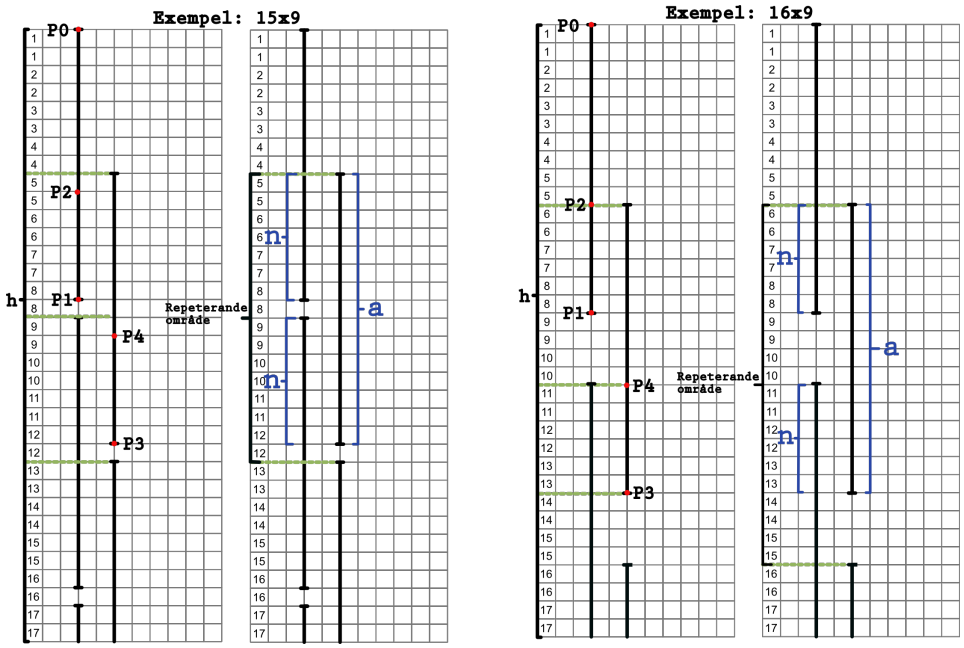
$$h_{max} = \frac{280 - 49}{5} = 46.2 \approx 46$$

Där 280 är antalet tillgängliga bRAM för arkitekturen, 49 är resurserna filtrena kräver och 5 är bredden på *data_memory.vhd*-blocket. Detta ger maxhöjden 46. Med dimensionerna 9x9 kommer inläsningen av varje minnesplats i genomsnitt ske 4.5 gånger för stride 1, samt 2.25 gånger för stride 2, se tabell 3.5. Genom att ändra dimensioner för blocket i höjdled kan antalet inläsningar för varje minnesplats reduceras. De grafiska beräkningarna medförde att ett matematiskt samband kan beskrivas.

Beräkningsmodellen illustreras i figur 3.3, där numreringen från 1-17, markerat med bokstaven h, symboliserar bildens höjd, (numreringen är endast en godtycklig representation och motsvarar inte de verkliga adresserna för vardera minnesplats). I det första exemplet i figur 3.3, är blockets dimensioner 15x9, där beräkningarna är gjorda för stride 1. Strecket, mellan punkterna P0 och P1, är 15 rutor högt och ritas från bildens första element markerat med punkt P0, detta motsvarar blockets höjd. På dessa 15 rutorna skall filtret, med höjden 7, appliceras fram till att strecket för blockets höjd tagit slut, markerat med punkten P1. Det första elementet som inte får plats inom filterområdet markeras med punkten P2. För att kunna göra max-pooling krävs det att punkten P2 hamnar mellan två likadana talpar. Skulle denna punkten skulle hamna mellan, till exempel, två fyror eller två femmor, måste den första fyran eller femman läsas in en gång till. I detta exemplet hamnar P2 mellan två femmor, vilket kräver den extra inläsningen, jämfört med exempel 16x9. Vid minneselementet ovanför punkten P2 ritas nästa streck, även detta strecket är 15 rutor långt vars slut markeras med P3. På samma sätt appliceras filtren för det nya strecket fram tills blockets höjd tagit slut. Det första elementet som inte får plats inom filterområdet markeras med punkten P4. Detta upprepas tills ett repeterande mönster uppstår. För att beräkna det genomsnittliga inläsningsantalet för exempel 15x9 och 16x9 i figur 3.3 används ekvation 3.2 och 3.3 som motiveras senare i avsnittet. Se följande beräkning:

$$M_{15} = \frac{15}{15-7} = 1.875$$

$$M_{16} = \frac{16}{16-6} = 1.6$$



Figur 3.3: Två grafiska exempel för att dimensionsbestämma blocket data_memory. Till vänser, ett exempel för 15x9 där punkterna P0-P4 är markerade för filterapplicering och överlappning, samt variablerna n och a för att visa beräkningsmetoden. På samma sätt till höger, ett exempel för 16x9 där samma markeringar är gjorda för att visa skillnaden i inläsningsantal.

Att utföra den grafiska lösningen från den minsta dimensionen till den största, är inte enbart tidskrävande, sannolikheten för att beräkningar blir fel är stor. En generell lösning för att hitta antalet inläsningar för större block utformades. Genom att använda föregående exempel, med dimensionerna 15x9, namngavs de olika sträckorna för kunna beskriva sambandet matematiskt, se figur 3.3. Sträckan a, är höjden på *data_memory.vhd* och n hur mycket varje inläsning överlappar. Antalet inläsningar per minnesplats, M, beskrevs nedan med ekvation 3.1, som utgick från det grafiska resonemanget i figur 3.3.

$$M = \frac{a + a - 2n + 2n}{2a - 2n} = \frac{a + a}{2a - 2n} = \frac{2a}{2a - 2n} = \frac{a}{a - n} \tag{3.1}$$

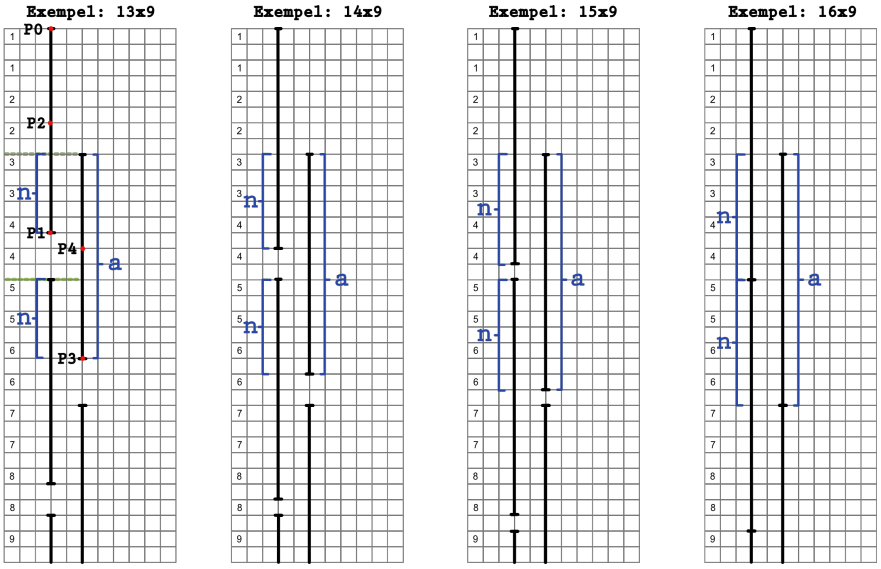
Skillnaden i överlappning beror på variabeln n. För stride 1 och för ojämna tal, i detta fallet 15, kommer n alltid vara lika med 7. Vid jämna tal kommer överlappningen n vara lika med 6. Med hjälp av ekvationerna nedan, 3.2 samt 3.3, kan inläsningsantalet beräknas för det önskade intervallet 9-46, se även tabell 3.5.

$$M_{jämn} = \frac{a}{a - n} \Bigg|_{n=6} \quad a \in \{10, 12, \ldots, 44, 46\} \tag{3.2}$$

$$M_{ojämn} = \frac{a}{a - n} \Bigg|_{n=7} \quad a \in \{9, 11, \ldots, 43, 45\} \tag{3.3}$$

3.1.2 Dimensionsberäkningar S2

Eftersom arkitekturen kommer att kunna köra stride 2, måste det genomsnittliga inläsningsantalet även beräknas för den möjligheten. Genom att utföra liknande grafiska beräkningar, som i tidigare exempel (se avsnitt 3.1.1), kan en generell lösning bestämmas. I figur 3.4, finns fyra grafiska exempel för stride 2, 13x9, 14x9, 15x9 och 16x9.



Figur 3.4: Fyra grafiska exempel för stride 2 för att dimensionsbestämma blocket data_memory. Från dimensionerna 13x9 till 16x9 där de olika beräkningsområdena är markerade med n och a. Punkterna P0-P4 är även markerade i första exemplet, 13x9. Detta för att visa skillnaden i inläsningsantal

Precis som för stride 1, behöver punkten P2 återigen hamna mellan två talpar. Återigen observeras överlappningen för vardera exempel i figur 3.4. I det första exemplet, 13x9, är a=13, och överlappningssträckan n=5. På samma sätt i exempel 14x9, är a=14 och överlappningssträckan n=6. För 15x9 är a=15 och n=16. I det sista exemplet 16x9, är a=6 och n=8. Detta kommer att vara det repeterande intervallet för överlappningen n, eftersom överlappningen för dimensionerna 12x9 är n=8, på samma sätt som 17x9 där n=5. Med hjälp av ekvation 3.1-3.3 , kan det genomsnittliga inläsningsantalet beskrivas för det önskade intervallet 9-46 för stride 2, se ekvation 3.4-3.7. Resultatet från följande beräkningar finns i tabell 3.5.

$$M_5 = \frac{a}{a-n}\Bigg|_{n=5} \quad a \in \{9, 13, \ldots, 41, 45\} \tag{3.4}$$

$$M_6 = \frac{a}{a-n}\Bigg|_{n=6} \quad a \in \{10, 14, \ldots, 42, 46\} \tag{3.5}$$

$$M_7 = \frac{a}{a-n}\Bigg|_{n=7} \quad a \in \{11, 15, \ldots, 39, 43\} \tag{3.6}$$

$$M_8 = \frac{a}{a-n}\Bigg|_{n=8} \quad a \in \{12, 16, \ldots, 40, 44\} \tag{3.7}$$

3.1.3 Resultat från dimensionsberäkningar

Beräkningsresultatet för det genomsnittliga inläsningsantalet för varje minnesplats visas i tabell 3.5. I projektet designades *data_memory.vhd* tidigt och resursförbrukningen för resterande funktionsblock var svår att förutse. Därför bestämdes dimensionerna till funktionsblocket med god marginal så att resterande funktionsblock i arkitekturen inte begränsades av resursförbrukningen för *data_memory.vhd*. Dimensionerna för blocket bestämdes till 18x9 vilket resulterade i det genomsnittliga inläsningsantalet för varje minnesplats till 1.5 för både stride 1 och stride 2. Med nuvarande arkitektur utan förbättringarna som diskuteras i avsnittet för vidareutveckling, se avsnitt 5.1, finns det resurser för att öka dimensionerna för blocket vilket hade minskat inläsningsantalet.

Stride 1:

a	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
n	7	6	7	6	7	6	7	6	7	6	7	6	7	6	7	6	7	6	7
M	4.5	2.5	2.75	2	2.17	1.75	1.88	1.6	1.7	1.5	1.58	1.43	1.5	1.38	1.44	1.33	1.39	1.3	1.35
a	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46
n	6	7	6	7	6	7	6	7	6	7	6	7	6	7	6	7	6	7	6
M	1.27	1.32	1.25	1.29	1.23	1.27	1.21	1.25	1.2	1.23	1.19	1.22	1.18	1.21	1.17	1.19	1.16	1.18	1.15

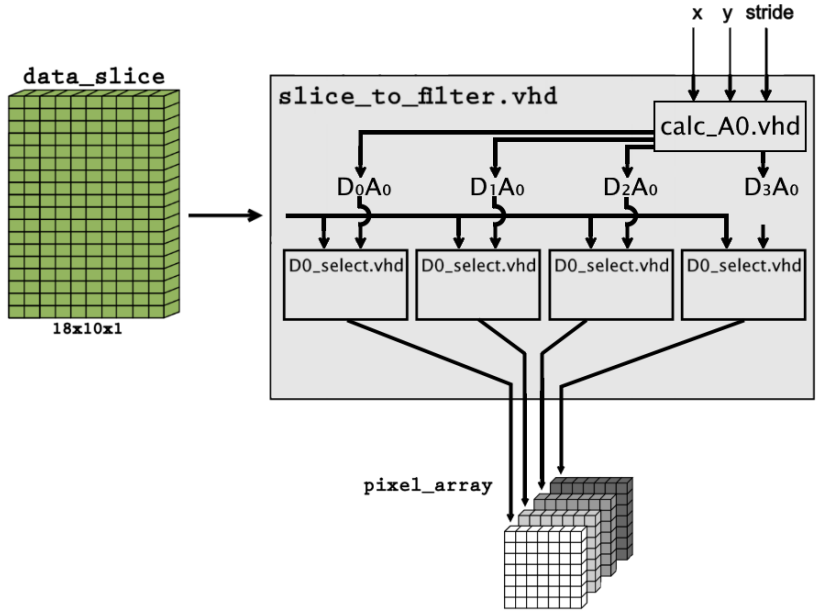
Stride 2:

a	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
n	5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8	5	6	7
M	2.25	2.50	2.75	3.00	1.63	1.75	1.88	2.00	1.42	1.50	1.58	1.67	1.31	1.38	1.44	1.50	1.25	1.30	1.35
a	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46
n	8	5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8	5	6
M	1.40	1.21	1.25	1.29	1.33	1.18	1.21	1.25	1.29	1.16	1.19	1.22	1.25	1.14	1.17	1.19	1.22	1.13	1.15

Tabell 3.5: Antalet inläsningar beroende på stride samt dimensioner på *data _memory.vhd*. Där a är blockets höjd, n är överlappningen för varje inläsning och M inläsningsantalet i genomsnitt.

3.2 Slice_to_filter

Slice_to_filter.vhd gör det möjligt att byta vilket område filtret appliceras på utan att överföra ny data utifrån. Detta genom att flytta filtret på området i data_slice med hjälp av x- och y-signalerna.



Figur 3.6: Utvidgad modell för *Slice_to_filter* blocket. Visar hur *Slice_to_filter* är uppdelad mellan *calc_A0* och *D0_select*.

Blocket *Slice_to_filter.vhd* tar emot en dataslice från *data_memory.vhd* och väljer vilka 4 områden som ska skickas till multiplikationsblocket. I varje dataslice finns flera möjliga platser att applicera filtret på. För att styra vilket arbetsområde som gäller används x- och y-signalerna från kontrollenheten samt s-signalen som indikerar vilken stride som ska användas för lagret. De områden som ska testas beror på vilken stride lagret använder enligt stridetabellerna nedan.

Stride = 1

x	0	1	0	1	0	1	0	1	0	1	0	1	0	1
y	0	0	1	1	2	2	3	3	4	4	5	5	6	6

Stride = 2

x	0	0	0	0
y	0	1	2	3

Ekvationerna i 3.8 använder koordinaterna x,y och stride värdet s och beräknas i *calc_A0* i figur 3.6 och ger indexet för översta vänstra hörnet för det område som ska hämtas ur dataslicen. *DO_select* använder indexet för att hitta de 49 värdena som ska skickas vidare.

$$\begin{aligned} D_0A_0 &= (x + 10y)s \\ D_1A_0 &= D_0A_0 + s \\ D_2A_0 &= D_0A_0 + 10s \\ D_3A_0 &= D_0A_0 + 11s \end{aligned} \tag{3.8}$$

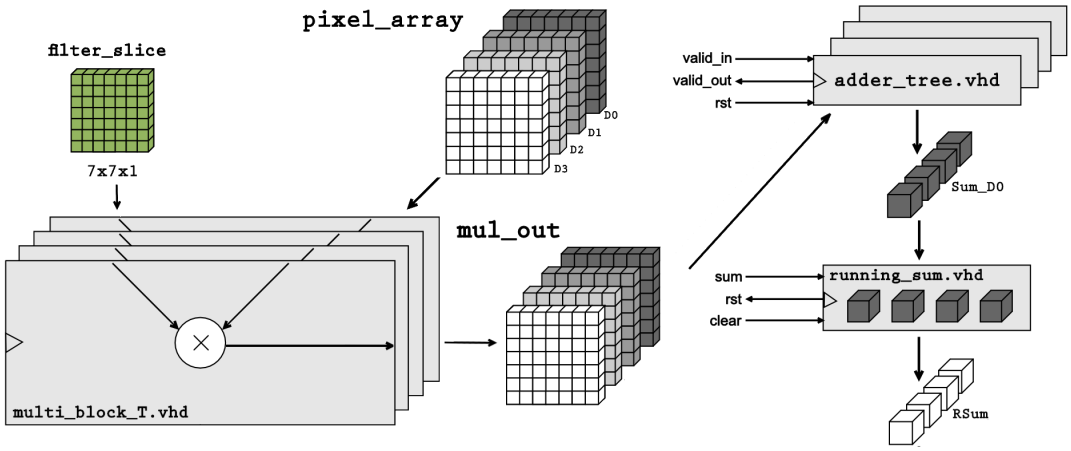
3.3 Filter

Filtermodulen består av ett tredimensionellt 7x7xN filter. För att ta sig igenom alla filter uppdateras filtret under körning. Detta är uppbyggt genom 49 inbyggda ramblock som har 1024 adresser som lagrar 8-bitars tal. Adressstorleken ger den över gränsen för djupet N av filtret.

3.3.1 Filter_Slice

En filterslice består av 49 värden som bestäms av den adress som skickas till filtret. Dessa värden kan alla läsas samtidigt, eftersom de kommer från olika ramblock och kan därför beräknas parallellt.

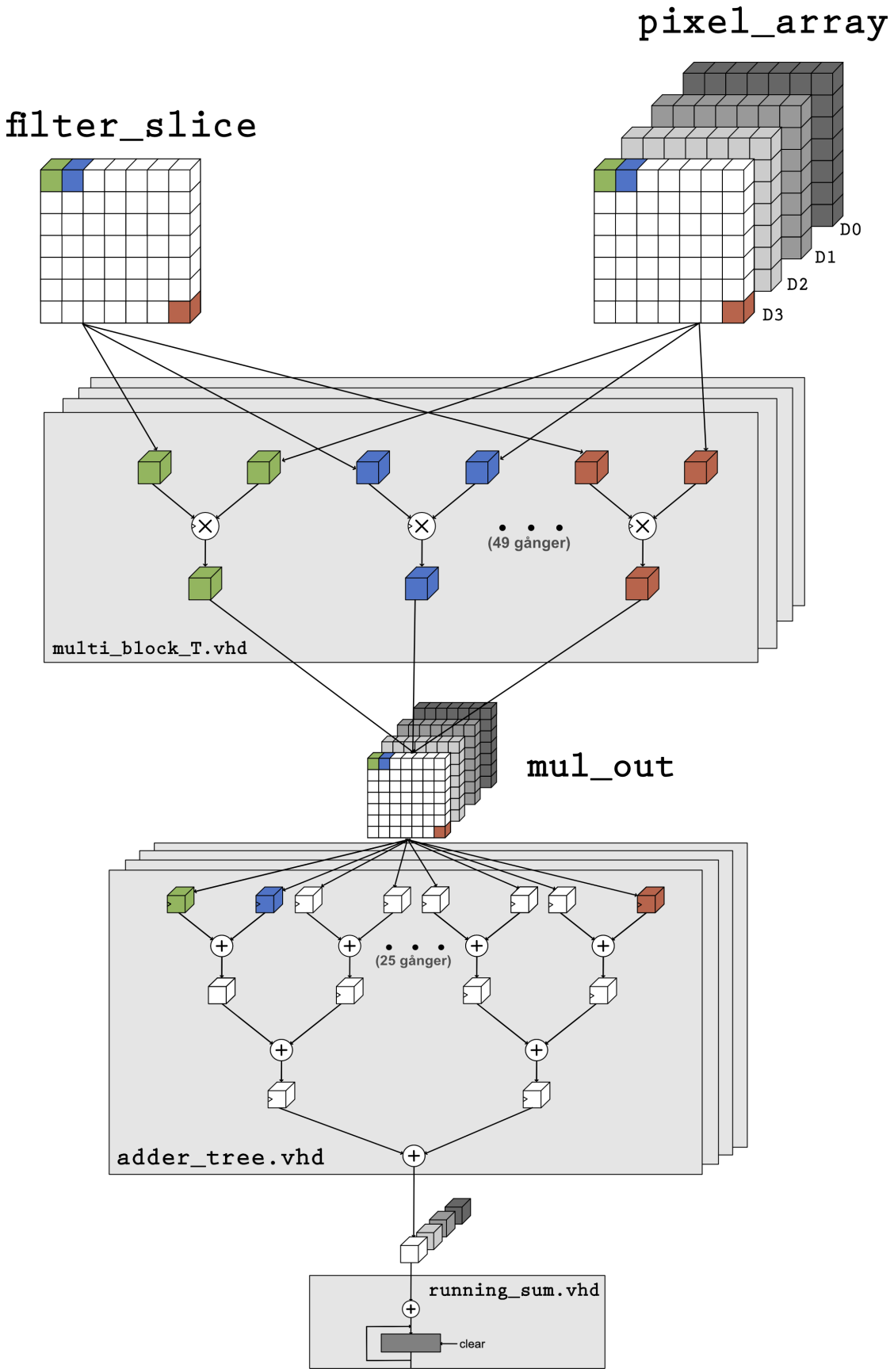
3.4 Beräkningsblock



Figur 3.7: Beräkningsblockets komponenter. Utdrag ur figur 3.1.

$$\sum_{i=0}^{f_{depth}} \overrightarrow{DataSlice}_i^K \bullet \overrightarrow{FilterSlice}_i = R_{sum}^K \tag{3.9}$$

Beräkningsblocket implementerar ekvationen 3.9 och består av *multi_block_T.vhd*, *adder_tree.vhd* och *running_sum.vhd* i figur 3.7 och 3.8. Den data som kommer från minnet och filtret multipliceras elementvis i *multi_block_T*. Detta utförs parallellt för alla fyra dataslicen då alla multipliceras med samma filterslice. Därefter konverteras listan med tal i *adder_tree* till ett stort binärt tal där alla tal ligger efter varandra för att matcha formatet till Brian Nezvadovitzs[8] automatiska adderarträd. Adderarträdet börjar summera när *valid in* signalen är hög och när summeringen är färdig ställs flaggan *valid out* hög i en klockcykel. Endast flaggan *valid out* från slice 0 skickas till kontrollenheten, vilket går att göra för att tiden för summeringen är konstant för alla fyra summerarträd. Flaggorna används för att summera över hela filtrer med hjälp av running_sum. När *valid out* flaggan är hög adderas summan till den rullande summan för alla slices av filtret. Kontrollsignalen *clear* rensar minnet när hela filtret är färdigt för att göra plats till nästa resultat. Summeringen kan ta in ett nytt tal varannan cykel även fast det tar ca 10 klockcykler för att göra hela summeringen.



Figur 3.8: Utvidgad modell av beräkningsblocket. Multiplikationer elementvis och parallellt. Additioner sker för alla lager samtidigt.

3.5 Aktiveringsfunktion

Activation.vhd implementerar ReLU-funktionen samt ser till att resultatet får plats i ett 8:a bitars tal med decimaltecknet på rätt plats. Detta genom att testa om MSB är ett vilket betyder att talet är negativt. Om talet är negativt tvingas resultatet till 0. Om talet är positivt avrundar den bort dom lägsta bitarna samt klipper bort de översta bitarna för att få ett 8-bitars tal.

3.6 Max-pooling

I *max_pool.vhd*, figur 3.1, implementeras max-pooling där de fyra resultaten jämförs och det största skickas vidare. Detta genom att först jämföra D0 med D1 samt D2 med D3 och i nästa klockcykel jämföra resultaten med varandra.

4
Resultat

Det slutgiltiga resultatet är en arkitektur för att utföra bildigenkänning med ett CNN anpassad efter utvecklingskortet *Zedboardens Zynq-7020*. I det här avsnittet kommer den arkitekturens funktionalitet beskrivas samt resultaten från de mätningar som utförts.

4.1 Prestanda

Arkitekturen uppnådde klockhastigheten 40 MHz på utvecklingskortet *Zedboardens Zynq-7020*. Att ta sig igenom en dataslice med ett djup på 127 tar 3612 klockcykler och motsvarar 348488 beräkningar enligt ekvation 4.1.

$$djup \cdot f_{size} \cdot n_{outputs} \cdot XY_{list} = 127 \cdot 49 \cdot 4 \cdot 14 = 348488 \tag{4.1}$$

Det genomsnittliga antalet beräkningar i sekunden är därför:

$$\frac{348488}{3612} \cdot 40 \cdot 10^6 \approx 3\ 860\ 000\ 000$$

I tabellen nedan jämförs resultatet med den teoretiska gränsen för Zedboarden och mot en GTX 1060 GPU. Den slutgiltiga beräkningskapaciteten ligger långt under FPGA:ns teoretiska-maximum som förväntat och även under grafikprocessorns beräkningskapacitet.

Teoretisk max	Resultat	GTX 1060 [9]	
667 MHz	40 MHz	1709 MHz	Klockhastighet
	177 mW	120W	Elförbrukning
293 GMACS	3.86 GMACS	68.36 GFLOPS	Beräkningskapacitet
	21,8 GMACS/W	0.570 GFLOPS/W	Energieffektivitet
220 st	196 st		DSP användning
140 st	69.5 st		bRAM användning
400 MBs		192.2 GBs	Överföringshastighet

4.1.1 Extrapolerat exempel

För att se hur resultatet ser ut på ett större exempel så har resultatet extrapolerats till en hel bild med dimensionerna i tabellen nedan.

bildstorlek (x,y,z)	filter	stride
$56 \cdot 56 \cdot 127$	512st	1

Tid för inläsning av data:
Tiden det tar att läsa in bilden och alla filter i det interna minnet är under 2 sekunder med den teoretiska max hastigheten på 400 MBs. Den mesta av den tiden går åt till att ladda in filtervikterna som går att se om man jämför med "Inläsningar utan vikter".

för varje position	för varje rad	totalt
$18 \cdot 127 \quad (data)$	$9 \cdot 18 \cdot 127(data)$	$5 \cdot rad$
$+512 \cdot 49 \cdot 127\ (filter)$	$+46 \cdot position$	

$$5 \cdot (9 \cdot 18 \cdot 127 + 46 \cdot (18 \cdot 127 + 512 \cdot 49 \cdot 127)) = 733\ 449\ 130$$

$$\frac{733\ 449\ 130\ inläsningar}{400\ 000\ 000/s} \approx 1.83\ sekunder$$

Beräkningstid:
Beräkningstiden är den långsammaste faktorn i den nuvarande implementationen. Det är också den som har de enklaste förbättringsmöjligheterna. Med alla förbättringsförslag skulle tiden minskas från 10.7 sekunder till 1.2 sekunder.

beräkningar för varje position	för varje rad	totalt
$14{\cdot}4 \cdot 49 \cdot 128 \cdot 512$	$46{\cdot}position$	$5 \cdot rad$

$$14 \cdot 49 \cdot 4 \cdot 128 \cdot 512 \cdot 46 \cdot 5 = 41\ 361\ 080\ 320$$

$$\frac{41\ 361\ 080\ 320\ beräkningar}{3\ 860\ 000\ 000\ beräkningar/s} \approx 10.7\ sekunder$$

Inläsningar utan vikter:
Om beräkningskapaciteten blir för hög så blir istället överföringshatigheten den begränsande faktorn. En lösning på det är att lagra mer filter i FPGA:n. Om inga filter behövs laddas över så går inläsningstiden ner från 1.83 sekunder till 0.0013 sekunder. Då blir processen begränsad av beräkningskapaciteten igen.

inläsningar för varje position	för varje rad	totalt
$18{\cdot}127$	$46{\cdot}position$	$5 \cdot rad$

$$18 \cdot 127 \cdot 46 \cdot 5 = 525\ 780$$

$$\frac{525\ 780\ inläsningar}{400\ 000\ 000\ inläsningar/s} \approx 0.0013\ sekunder$$

5
Slutsats och diskussion

Arbetet har resulterat i en arkitektur som är energieffektiv och använder en stor del av de tillgängliga resurserna. Framförallt DSP-modulerna där 196 av 220 stycken används. Beräkningskapaciteten är betydligt lägre än den teoretiska max gränsen. Detta beror på den låga klockhastigheten och att beräkningskapaciteten är begränsad av summeringsträdet vilken bara kan ta emot nya värden varannan klockcykel.

Trots begränsningarna är resultatet mer energieffektivt än en konsument-GPU. Vilket tyder på att FPGA:er kan användas för små nätverk i lågenergiapplikationer, exempelvis drönare. Det finns förbättringar vilka omnämns i nästkommande kapitel 5.1, där beräkningskapaciteten uppskattas till 34 GMACS. Detta motsvarar halva beräkningskapaciteten hos en GTX 1060. CNN-arkitekturen är väl anpassad till FPGA:er då samma filter återanvänds flera gånger vilket begränsar mängden data som behöver överföras till FPGA:n. För att få ut det mesta ur en FPGA som accelerator bör den specifika FPGA:n väljas så att alla filter får plats i minnes och har så många DSP:er som möjligt.

5.1 Vidareutveckling

Med relativt små förändringar kan beräkningskapaciteten öka dramatiskt. Ett annat summeringsträd skulle kunna ta emot värden varje klockcykel och på så sätt dubbla bräkningskapaciteten. Detta skulle medföra att alla andra komponenter i beräkningskedjan kan ta emot nya värden varje klockcykel. De DSP:er som finns är inte utnyttjade till fullo då alla multiplikationer är 8-bitar. De kan konfigureras till att göra två multiplikationer samtidigt med 8-bitars tal vilket skulle dubbla beräkningskapaciteten igen [10]. Även klockhastigheten går att förbättra genom att minska den kritiska vägen som i nuläget ligger mellan *data_slice* och multiblocket.

Referenser

[1] Karim, Shahid and Zhang, Ye and Laghari, Asif *Image Processing Based Proposed Drone for Detecting and Controlling Street Crimes* 2017
Avalable: https://ieeexplore.ieee.org/document/8359925 [Använd 20190608]

[2] Synective Labs *FPGA-related services.* [Online].
Available: https://synective.se/about-us/ [Använd 20190519]

[3] Vivado Design Suite *Design- och programmeringsverktyg*[Online].
Available:

[4] J. G. Webster, *Medical Instrumentation, Application and Design*, Fourth Edition.
John Wiley Sons, inc,2009.

[5] Wikimedia Commons *Figur, de fyra olika faserna - biologisk nervcell.* [Online].
Available:https://upload.wikimedia.org/wikipedia/commons/4/4a/Action_potential.svg
[Hämtad 20190527]

[6] S, Eger. P, Youssef och I, Gurevych. *Is it Time to Swish? Comparing Deep Learning Activation Functions Across NLP tasks* 2019. [Online].
Available: https://aclweb.org/anthology/D18-1472 [Använd 20190508]

[7] GitHub *Figur, exempelbider för filter.* [Online].
Available:https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/ [Hämtad 20190608]

[8] Brian Nezvadovitz *vhdl-examples.* [Online].
Available:https://github.com/khaledhassan/vhdl-examples/tree/master/adder$_tree$[Hämtad 20190527] , GitHub repository

[9] Techpowerup *GPU database.* [Online].
Available:https://www.techpowerup.com/gpu-specs/geforce-gtx-1060-6-gb.c2862[använd 20190519]

[10] Fu, Yao and Wu, Ephrem and Sirasao, Ashish and Attia, Sedny and Khan, Kamran and Wittig, Ralph *Deep learning with int8 optimization on xilinx devices* White Paper 2016.