UNIVERSITATEA SAPIENTIA DIN CLUJ-NAPOCA
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE, TÎRGU-MUREȘ
SPECIALIZAREA CALCULATOARE

Implementarea rețelelor neuronale artificiale prin sinteză de nivel înalt.

PROIECT DE DIPLOMĂ

Coordonator ştiinţific:
Conf. dr. ing. Brassai Sándor Tihamér

Absolvent:
Bustya Balázs

2021

UNIVERSITATEA "SAPIENTIA" din CLUJ-NAPOCA **Viza facultății:**
Facultatea de Științe Tehnice și Umaniste din Târgu Mureș
Specializarea: **Calculatoare**

LUCRARE DE DIPLOMĂ

Coordonator științific:	Candidat: **Bustya Balázs**
conf. dr. ing. Brassai Sándor Tihamér	Anul absolvirii: **2021**

a) Tema lucrării de licență:

Implementarea rețelelor neuronale artificiale prin sinteză de nivel înalt.

b) Problemele principale tratate:

- Studiu bibliografic privind implementarea aplicațiilor prin sinteză de nivel înalt
- Frameworkul LeFlow pentru implementarea rețelelor neuronale artificiale în circuite FPGA
- Implementarea funcțiilor de activare în hardware

c) Desene obligatorii:

- Diagrama de flux pentru frameworkul LeFlow
- Diagrama de flux pentru generarea în hardware a rețelei neuronale artificiale
- Interfațarea în System Generator a rețelei neuronale artificiale implementate în hardware
- Schema bloc a sistemului de interfațare a rețelei neuronale cu procesorul ARM implementată în Xilinx Vivado
- Diagrame UML privind software-ul realizat.
- Compararea diferitelor microarehitecturi pentru produsul vectorial obținute cu diferite directive de optimizare
- Măsurători realizate pentru testarea sistemului implementat și compararea rezultatelor obținute

d) Softuri obligatorii:

- Soft în în Vivado HLS pentru implementarea rețelelor neuronale artificiale prin sinteză de nivel înalt
- Modele System Generator pentru interfațarea rețelei neuronale
- Script .tcl pentru automatizarea procesului de generare a rețelei neuronale

e) Bibliografia recomandată:

[1] Shymkovych, V., Telenyk, S. and Kravets, P., 2021. Hardware implementation of radial-basis neural networks with Gaussian activation functions on FPGA. Neural Computing and Applications, pp.1-13.
[2] Noronha, D.H., Gibson, K., Salehpour, B. and Wilton, S.J., 2018, December. Leflow: Automatic compilation of tensorflow machine learning applications to fpgas. In 2018 International Conference on Field-Programmable Technology (FPT) (pp. 393-396). IEEE.
[3] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang and J. Cong, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE, 2017, pp. 152-159

f) Termene obligatorii de consultații: săptămânal

g) Locul și durata practicii: Universitatea Sapientia,
Facultatea de Științe Tehnice și Umaniste din Târgu Mureș

Primit tema la data de: 31.03.2020
Termen de predare: 06.07.2019

Semnătura Director Departament **Semnătura coordonatorului**

Semnătura responsabilului programului de studiu **Semnătura candidatului**

Model tip a.

Declaraţie

Subsemnatul **Bustya Balázs**, absolvent al/a specializării **Calculatoare**, promoţia **2021** cunoscând prevederile Legii Educaţiei Naţionale 1/2011 şi a Codului de etică şi deontologie profesională a Universităţii Sapientia cu privire la furt intelectual declar pe propria răspundere că prezenta lucrare de licenţă/proiect de diplomă/disertaţie se bazează pe activitatea personală, cercetarea/proiectarea este efectuată de mine, informaţiile şi datele preluate din literatura de specialitate sunt citate în mod corespunzător.

Localitatea, Târgu Mureș

Data: 02.07.2021

Absolvent

Semnătura.....Bustya.....

Ide kerül a Turnitin similarity report

Implementarea rețelelor neuronale artificiale prin sinteză de nivel înalt.

Extras

Rețelele neuronale artificiale au obținut o mare popularitate prin acuratețea lor în procesarea imaginilor, extragerea datelor, sarcini de clasificare și sisteme de luare a deciziilor, în care anterior doar oamenii erau capabili să ia deciziile corecte. Cu toate acestea, rețelele neuronale nu pot încă înlocui complet oamenii, deoarece oamenii le proiectează în funcție de sarcina dată. Acceleratoarele hardware sunt utilizate pentru a crește performanța rețelelor neuronale. În ultimii ani au existat numeroase studii privind acceleratoarele hardware. O astfel de opțiune sunt circuitele FPGA. Principalele avantaje ale FPGA sunt paralelismul, consumul redus de energie și flexibilitatea. Implementarea rețelelor neuronale pe FPGA este o sarcină dificilă și consumatoare de timp, care poate fi efectuată doar de profesioniști cu experiență. Cadrul pe care l-am dezvoltat oferă soluție la această problemă. Scopul cadrului este de a accelera și a facilita implementarea rețelelor neuronale care ”rulează” pe un circuit FPGA, de a accelera rețelele neuronale, de a studia opțiunile de optimizare în ceea ce privește dimensiunea și performanța și de a automatiza procesul de programare. Modelele de rețele neuronale parametrizabile, implementate prin sinteza la nivel înalt discutată în disertație, servesc ca bază a cadrului, care generează codul rețelelor neuronale modelate în C++ optimizate prin directive. Instrumentul Vivado HLS generează nucleul IP al rețelei neuronale din acest cod. Se integrează automat într-un sistem încorporat în Vivado. Instrumentul Vivado generează fișierul de biți necesar pentru programarea FPGA și programează circuitul. Disertația include implementarea unei rețele neuronale perceptron multistrat și a unei rețele de funcții de bază radială, implementări hardware ale funcțiilor de activare neliniare, compararea reprezentărilor numerice în virgulă fixă și virgulă mobilă.

Cuvinte cheie: rețele neuronale, FPGA, cadru, sinteză la nivel înalt, MLP, RBF

SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR
SZÁMÍTÁSTECHNIKA SZAK

Neuronhálók FPGA alapú megvalósítása magas szintű szintézissel.

DIPLOMADOLGOZAT

Témavezető:
Dr. Brassai Sándor Tihamér

Végzős hallgató:
Bustya Balázs

2021

Kivonat

A mesterséges neuronhálók nagy népszerűséget szereztek pontosságuknak köszönhetően képfeldolgozásban, adatbányászatban, osztályozási feladatokban, valamint olyan döntéshozó rendszerekben, amelyekben korábban csak az ember volt képes helyes döntéseket hozni. Ugyanakkor a neuronhálók még nem tudják teljesen helyettesíteni az embert, mivel az emberek tervezik meg őket az elvégzendő feladat függvényében. A neuronhálók teljesítményének növelése érdekében hardveres gyorsítókat alkalmaznak. Az utóbbi években rengeteg tanulmány készült hardveres gyorsítókról. Az egyik ilyen lehetőség az FPGA áramkörök, melyek legfőbb előnyei a párhuzamosíthatóság, alacsony fogyasztás és rugalmasság. Azonban a neuronhálók megvalósítása FPGA-n igencsak időigényes és nehéz feladat, melyet csak tapasztalt szakemberek tudnak elvégezni. Erre a problémára nyújt megoldást az általunk fejlesztett keretrendszer neuronhálók megvalósítására. A keretrendszer célja az FPGA-n "futó" neuronhálók megvalósításának felgyorsítása, megkönnyítése, a neuronhálók gyorsítása, optimalizálási lehetőségek tanulmányozása méret és teljesítmény szempontjából, valamint a felprogramozási folyamat automatizálása. A dolgozatban tárgyalt magas szintű szintézissel megvalósított, paraméterezhető neuronháló modellek alapjául szolgálnak a keretrendszernek, amely a C++-ban modellezett neuronhálók kódját generálja direktívákkal optimalizálva. A Vivado HLS eszköz ebből a kódból kigenerálja a neuronháló IP magját. Ez automatikusan illesztődik egy Vivadoban kiépített rendszerbe. A Vivado eszköz generálja az FPGA programozásához szükséges bit fájlt és felprogramozza az áramkört. A dolgozat tartalmazza egy többrétegű perceptron típusú neuronháló, valamint egy radiális bázisfüggvényekből álló hálózat megvalósítását, nemlineáris aktivációs függvények hardveres megvalósításait, fixpontos és lebegőpontos számábrázolások összehasonlítását.

Kulcsszavak: neuronhálók, FPGA áramkör, keretrendszer, magas szintű szintézis, MLP, RBF

Abstract

Artificial neural networks have gained great popularity for their accuracy in image processing, data mining, classification tasks, and decision-making systems in which previously only humans were able to make the right decisions. However, neural networks cannot yet completely replace humans, because humans design them depending on the given task. Hardware accelerators are used to increase the performance of neural networks. There have been plenty of studies on hardware accelerators in recent years. One of the options are FPGA circuits. The main advantages of FPGAs used for hardware acceleration are parallelism, low power consumption and flexibility. Implementing neural networks on FPGA is a very time consuming and difficult task, that can only be performed by experienced professionals. The framework we have developed provides solution to this problem. The purpose of the framework is to accelerate and facilitate the implementation of neural networks "running" on an FPGA, to accelerate neural networks, to study optimization options in terms of size and performance, and to automate the programming process. The parameterizable neural network models, implemented by the high-level synthesis discussed in the dissertation, serve as the basis of the framework, that generates the code of the neural networks modelled in C ++ optimized by directives. The Vivado HLS tool generates the IP core of the neural network from this code. It automatically integrates into a system built in Vivado. The Vivado tool generates the bit file needed to program the FPGA and programs the circuit. The paper includes the implementation of a multilayer perceptron neural network and a radial basis function network, hardware implementations of nonlinear activation functions, comparison of fixed-point and floating-point number representations.

Keywords: neural networks, FPGA , framework, high-level synthesis, MLP, RBF

Tartalomjegyzék

Ábrák, táblázatok jegyzéke 10
1. Bevezető 12
1.1. Célkitűzések 13
2. Irodalmi áttekintés 14
2.1. Magas szintű szintézis (HLS) 14
2.1.1. Kompilálás (Compilation) 14
2.1.2. Erőforrás kiosztás (Allocation) 15
2.1.3. Ütemezés 15
2.1.4. Összekapcsolás (Binding) 15
2.1.5. RTL kód generálása (Generation) 15
2.1.6. Vivado HLS 16
2.1.7. HLS előnyei RTL szintű tervezéshez képest 19
2.1.8. HLS hátrányai RTL szintű tervezéshez képest 19
2.2. Hasonló keretrendszer - LeFlow 19
3. A rendszer specifikációi és architektúrája 21
3.1. Felhasználói követelmények 21
3.2. Funkcionális követelmények 21
3.3. Nem Funkcionális követelmények 21
3.4. A rendszer architektúrája 22
4. Megvalósítás 24
4.1. Módszerek 24
4.2. Többrétegű perceptron típusú háló 25
4.2.1. Az inger kiszámolása 25
4.2.2. Egy réteg felépítése 32
4.2.3. Szigmoid aktivációs függvény szakaszonkénti lineáris közelítéssel 34
4.2.4. Szigmoid aktivációs függvény keresőtáblázattal 34
4.2.5. Egész alapú számábrázolás használata 35
4.3. Fix pontos számábrázolás lebegőpontos számábrázolással szemben 37
4.3.1. Lebegőpontos számábrázolás 38
4.3.2. Előjeles fixpontos<32,16> 39
4.3.3. Előjeles fixpontos<16,8> 39
4.3.4. Előjeles fixpontos<12,6> 40
4.3.5. Mérések összefoglalása, kiértékelése 41
4.4. Radiális bázisfüggvényekből álló hálózat 41
4.4.1. Távolság (norma számolás) 42
4.4.2. Gauss bázisfüggvény 43
4.4.3. Szakaszonkénti lineárs megközelítés és keresőtáblázat módszerek összehasonlítása 44
4.5. Modell optimalizálása 48
4.6. A neuronháló interfészelése 48
4.7. Hibakeresés (Debuggolás) 50

4.8. A rendszer automatizálása 52
4.9. Felmerült problémák és megoldásaik 54
5. Megoldott feladat 54
6. Eredmények 57
6.1. Hasonló rendszerekkel való összehasonlítás 58
6.1.1. Saját rendszer összehasonlítva LeFlow-al 58
6.1.2. A CPU-n futatott python modell összehasonlítása az FPGA-n futatott modellel 59
7. Következtetések 59
7.1. Megvalósítások 59
7.2. További célkitűzések 60
7.3. Köszönetnyilvánítás 61
8. Irodalomjegyzék 61

Ábrák, táblázatok jegyzéke

1. ábra: HLS alapú hardver kialakításának folyamata [5] 16
2. ábra: LeFlow keretrendszer folyamatábrája. 20
3. ábra: Rendszer folyamatábra. 22
4. ábra: Simulink-ben System Generator-al felépített rendszer. 23
5. ábra: Vivado-ban felépített hardver tömbvázlata. 24
6. ábra: Vektor szorzás direktívákkal. 25
7. ábra: Mérés direktívák használata nélkül. 26
8. ábra: Erőforrás felhasználás statisztika, direktívák használata nélkül. 26
9. ábra: Vektor szorzás mikróarchitektúra direktívák alkalmazása nélkül. 26
10. ábra: Mérés, vektor szorzás kiterjesztve 4-es faktorral. 27
11. ábra: Erőforrás felhasználási statisztika, vektor szorzás kiterjesztve 4-es faktorral. 28
12. ábra: Mérés, tömbök particionálva. 28
13. ábra: Vektor szorzás mikróarchitektúra, ciklus kiterjesztve, tömbök partícionálva. 29
14. ábra: Mérés, vektor szorzás csővezetékezve. 30
15. ábra: Erőforrás felhasználási statisztika, vektor szorzás csővezetékesítve. 30
16. ábra: Vektor szorzás, ciklus kiterjesztve, tömbök partícionálva, ciklus csővezetékesítve. 31
17. ábra: Egy neuron réteg. 32
18. ábra: Egy neuron. 32
19. ábra: Mérés, neuronok párhuzamosítva. 33
20. ábra: Erőforrás felhasználási statisztika, neuronok párhuzamosítva. 33
21. ábra: Neuronok kiterjesztve (párhuzamosítva) mikróarchitektúra. 33
22. ábra: Fa típusú architektúra. 35
23. ábra: Erőforrás felhasználási statisztika, inger kiszámolása fa típusú architektúrával, neuronok csővezetékesítve. 36
24. ábra: Mérés, integer műveletekkel, vektor szorzás kiterjesztve, neuronok csővezetékesítve 37
25. ábra: 6 elemre generált fa típusú vektorszorzás architektúra. 37
26. ábra: Mérés, 4 bájtos lebegőpontos adattípust használva. 38
27. ábra: Erőforrás felhasználási statisztika, 4 bájtos lebegőpontos adattípust használva. 38
28. ábra: Mérés, fixpontos<32,16> adattípust használva. 39
29. ábra: Erőforrás felhasználási statisztika, fixpontos<32,16> adattípust használva. 39

30. ábra: Mérés, fixpontos<16,8> adattípust használva. 40
31. ábra: Erőforrás felhasználási statisztika, fixpontos<16,8> adattípust használva. 40
32. ábra: Erőforrás felhasználási statisztika, fixpontos<12,6> adattípust használva. 40
33. ábra: Fixpontos és lebegő pontos számábrázolás összehasonlítása [17]. 41
34. ábra: RBF neurális háló. 42
35. ábra: Távolság számolás. 42
36. ábra: Szakaszonkénti lineáris megközelítés. 43
37. ábra: Keresőtáblázat 44
38. ábra: Lineáris szakaszonkénti megközelítéssel 45
39. ábra: Keresőtáblázattal 45
40. ábra: Keresőtáblázattal 46
41. ábra: Lineáris szakaszonkénti megközelítéssel 46
42. ábra: Lineáris szakaszonkénti megközelítéssel 46
43. ábra: Keresőtáblázattal 46
44. ábra: Lineáris szakaszonkénti megközelítéssel 47
45. ábra: Keresőtáblázat 47
46. ábra: A modell korlátai. 48
47. ábra: Neuronháló interfészelés. 49
48. ábra: HLS interfész kialakítása direktívákkal. 49
49. ábra: Hibakeresés tevékenység diagram 51
50. ábra: A neuronháló bemenet beolvasásának ellenőrzése ILA-val 51
51. ábra: Tcl szkript a Vivado HLS feladatainak automatizálására. 53
52. ábra: Tcl szkript a Vivado feladatainak automatizálására. 54
53. ábra: HLS erőforrás használat és teljesítmény becslés (35->10) 55
54. ábra: HLS erőforrás használat és teljesítmény becslés (35->56->10). 56
55. ábra: HLS erőforrás használat és teljesítmény becslés (35->128->64->32->10) 56

1. táblázat: Lineáris szakaszonkénti megközelítés és keresőtáblázat összehasonlítás 44
2. táblázat: A modellek futási ideje 57
3. táblázat: A modellek tanítási és futási ideje a [18] dolgozatból. 57
4. táblázat: A modellek négyzetes hibáinak az átlaga a [18] dolgozatból. 58

1. Bevezető

A természetes intelligencia megértésére és intelligens gépek építésére régóta megszületett a törekvés az emberben, már a görög mitológiában van szó mechanikus emberről. Ennek a törekvésnek a révén jött létre a mesterséges intelligencia. A legtöbb típusú mesterséges neuronháló szerkezete nagyban hasonlít a gerincesek idegrendszeréhez és azok több tulajdonságát is modellezik. Például a neuronok szinapszisokon keresztül kommunikálnak egymással [1].

Napjainkban mesterséges neurális hálózatok nagy érdeklődésnek örvendenek a kutatások és ipar szférájában, főleg a gépi látás és mesterséges intelligencia területén mutatott pontosságuknak köszönhetően. Azonban ezek a rendszerek hatalmas számítási kapacitást igényelnek [1].

A GPU-k (grafikus feldolgozó egység) rendelkeznek a szükséges számítási kapacitással, viszont nem elég energia takarékosak a beágyazott rendszerekbe történő alkalmazáshoz. A CPU-k (központi feldolgozó egység) párhuzamosítási lehetőségei korlátozottak és szintén nem elég energia takarékosak. Az ASIC (alkalmazás specifikus integrált áramkör) alkalmas lehet energia takarékosság és teljesítmény szempontjából is, de nagy hátránya, hogy nem újrakonfigurálható, nem rugalmas egy új architektúra kialakítására. [2].

Egyik alternatíva a neuronhálók beágyazott rendszerekben történő megvalósításához az FPGA (programozható kapumátrix) áramkör. Logikai blokkokból épül fel, amelyeket egyszerű funkcionalitásra programozhatunk. A különböző logikai blokkokat össze lehet kapcsolni egymással az FPGA-ban található kapcsoló mátrixok segítségével, ezáltal kialakítva bonyolultabb szerkezeteket. Egy FPGA lapon több egymástól független struktúra hozható létre, amelyek párhuzamosan dolgoznak. Így alkalmas a neurális hálózatokban jelen levő modularitás és párhuzamos végrehajtás megvalósítására. Az újraprogramozhatóság következtében különböző modelleket valósíthatunk meg ugyanazon az FPGA áramkörön.

Az FPGA áramkörök egy nagyságrenddel kevesebb energiát fogyasztanak a grafikus feldolgozó egységeknél és jelentősen gyorsabbak az általános célú processzoroknál. Rugalmasak és alkalmasak fejlesztésre újrakonfigurálhatóságuknak köszönhetően. A rendszer karbantartható, valamint bármely ötlet azonnal megvalósítható, az alkalmazás specifikus áramkörökkel szemben, ahol megkell várnunk a hosszú tervezési majd gyártási folyamatot.

Az FPGA áramkörök tartják a lépést a technikai fejlődéssel. A legújabb FPGA lapok már 7 nm-es tranzisztor mérettel készülnek, aminek köszönhetően még kisebbek és nagyobb teljesítményűek.

Méretüknek és alacsony fogyasztásuknak köszönhetően jól alkalmazhatóak beágyazott rendszerekben.

Hátrányaik közé tartozik, hogy korlátozottak az erőforrásaik akárcsak a GPU-knak, és lassú a fordítás. További hátrányuk, hogy az FPGA-n történő fejlesztéshez szükséges a VHDL vagy Verilog programozási nyelvek ismerete, valamint digitális elektronika alapismeretek. Az FPGA áramköröket programozó mérnökök kell tudják használni a fejlesztési és szimulációs eszközöket. Tehát az FPGA-n történő fejlesztés sok tapasztalatot szakértelmet és időt igényel. A neuronhálók pedig gyorsan fejlődnek, különböző feladatokhoz különböző felépítésű neuronhálóra van szükség. Ezek állandó újratervezése, új architektúrák létrehozása FPGA-ra nagyon hatástalan lenne [3]. Erre a problémára nyújtana megoldást az általunk kiépített keretrendszer.

1.1. Célkitűzések

Kutatásunk célja egy keretrendszer megvalósítása, mely lehetővé teszi különböző típusú és méretű neurális hálózatok FPGA áramkörön történő modellezését, a felhasználó által megadott paraméterekből történő megépítését és gyorsítását.

A kutatás céljai közé tartozik a hardver megvalósítású neurális hálózatok optimalizálási lehetőségeinek tanulmányozása gyorsítás és erőforrás felhasználás szempontjából. A keretrendszer lehetőséget biztosítana a felhasználónak, hogy adott korlátok között meghatározza a gyorsítás és a felhasznált erőforrás arányát. A neurális hálókat szeretnénk egyaránt tesztelni és tanítani is az FPGA-n.

A kutatás során magas szintű szintézis (HLS) van alkalmazva a neuronháló hardver alapú megvalósítására A HLS elemez egy algoritmikus leírást (C, C++, SystemC, Matlab) és kigenerálja belőle a hardvert. A hardver implementációt direktívák segítségével tudjuk befolyásolni. Előnye a gyorsított fejlesztési és tesztelési folyamat (akár 10-szer gyorsabb fejlesztés, mint VHDL-ben). Hátránya, hogy a kisebb mértékben tudjuk befolyásolni a kigenerált, ezért általában kisebb hatékonyság érhető el, mint VHDL használatával.

Kutatásom várható eredményei paraméterezhető neuronhálók C++-ban történő tervezése és implementációja, direktívákkal ellátva a HLS eszköz számára. A neuronhálók különböző hardver mikróarchitektúrákkal történő megvalósítása és azok összehasonlítása. Különböző nemlineáris aktivációs függvények hardver megvalósítása többféle módszerrel és azok összehasonlítása. Az általunk épített keretrendszer összehasonlítása más már létező hasonló keretrendszerekkel (LeFlow) és neuronhálók hardvere megvalósításaival. A már létező keretrendszerek

hiányosságainak feltárása és ezek beépítése a saját keretrendszerünkbe, vagy a saját keretrendszer beépítése más keretrendszerbe úgy, hogy pótolja annak a hiányosságait.

Céljaim közé tartozik a rendszer használatának részleges vagy teljes automatizálása tcl szkriptekkel.

Megvalósítandó neuronháló típusok és aktivációs függvények: többrétegű perceptron típusú háló szigmoid aktivációs függvénnyel, radiális bázisfüggvényekből álló háló Gauss aktivácós függvénnyel.

2. Irodalmi áttekintés

2.1. Magas szintű szintézis (HLS)

A HLS eszköz bemenete az alkalmazás funkcionális leírása kiegészítve a kényszerfeltételekkel (direktívákkal) valamint, könyvtárelemek az elérhető komponensekkel (tetszőleges pontosságú adattípusok, DSP modulok, memória modulok). Kimenete a regiszter szintű leírás (RTL). A tervező leírja a modulok funkcionalitását és megtervezi a modulok közötti kommunikációt. A HLS eszköz kompilálja a funkcionális leírást, kiosztja a hardver erőforrásokat, ütemezi a műveleteket órajel szerint, a műveleteket funkcionális egységekhez köti, a változókat tároló elemekhez kapcsolja és generálja az RTL architektúrát [4].

2.1.1. Kompilálás (Compilation)

Első lépésként a HLS a bementként kapott kód kompilálását végzi el. Ez a folyamat magába foglalja a kód optimalizálását. Eltávolítja az úgynevezett „dead-code”-ot, vagyis azokat a részeket, amelyeknek az eredménye nem használódik fel. Eltávolítja a hamis adatfüggőségeket, kiértékeli a konstans kifejezéseket (constant folding) és ciklus átalakításokat (loop transformation) végez, hogy felfedje a párhuzamosítható részeket. A kompilátor által generált modellből kimutathatóak az adat- és kontrollfüggőségek. Az adatfüggőségek ábrázolhatóak egy adatfolyam gráfként (DFG), amiben minden csúcs egy műveletnek felel meg és a csomópontok közötti utak jelentik a bemeneteket, kimeneteket és ideiglenes változókat. Ez a gráf létrehozható, ha töröljük a kontrollfüggőségeket a leírásból a kompilálás során. Ennek az eléréséhez teljesen kikell terjeszteni a ciklusokat, ami nagy erőforrásigényű modellhez vezethet és csak statikus ciklus számláló esetén valósítható meg. Az adatfolyam gráfot kibővítve a kontroll függőségekkel kapunk egy irányított gráfot, ahol az élek jelentik a vezérlő folyamot. A csúcsok alap blokkokat jelentenek, amik nem tartalmaznak elágazásokat, csak szekvenciális állításokból épülnek fel. Ezzel a gráffal

ábrázolhatóak ismeretlen ciklusszámlálóval rendelkező ciklusok, de a párhuzamosíthatóság csak az alap blokkokon belül egyértelműsíthető. Az alap blokkok közötti párhuzamosíthatóság felfedéséhez további elemzések lennének szükségesek. Ezek lehetnek automatikusok, vagy manuálisan megadhatóak a felhasználó által [4].

2.1.2. Erőforrás kiosztás (Allocation)

Második lépésként a HLS eszköz meghatározza a szükséges erőforrásokat. HLS eszköztől függően néhány komponens kiválasztható az ütemezés, vagy a műveletek funkcionális egységhez való kötésénél. Az erőforrásokat az RTL komponens könyvtárból választja ki. A könyvtárnak tartalmaznia kell a komponensek tulajdonságait (késleltetés, teljesítmény stb.). Minden művelethez kiválaszt legalább egy komponenst.

2.1.3. Ütemezés

Harmadik lépés az ütemezés. A műveletek végrehajthatóak 1 vagy több órajel alatt, a hozzájuk rendelt funkcionális egységtől függően. A műveletek egymás után láncolhatóak, vagyis egyik művelet eredménye a másik bemenetét képezi és párhuzamosíthatóak, ha nincs közöttük adattfüggőség és elegendő erőforrás áll rendelkezésre.

2.1.4. Összekapcsolás (Binding)

Negyedik lépés a változók és műveletek funkcionális egységekhez való kötése. Minden változót egy tároló egységhez kell kapcsolni. Azok a változók, amelyek nem fedik át egymás élettartamát köthetők ugyanahhoz az erőforráshoz. Minden műveletet egy olyan funkcionális egységhez kell kötni, amelyik képes elvégezni azt. Ha több ilyen funkcionális egység is rendelkezésre áll, akkor a kötést végző algoritmus kell kiválassza az optimálist. A tárolókat össze kell kapcsolni a funkcionális egységekkel, például síneken vagy multiplexereken keresztül [4].

2.1.5. RTL kód generálása (Generation)

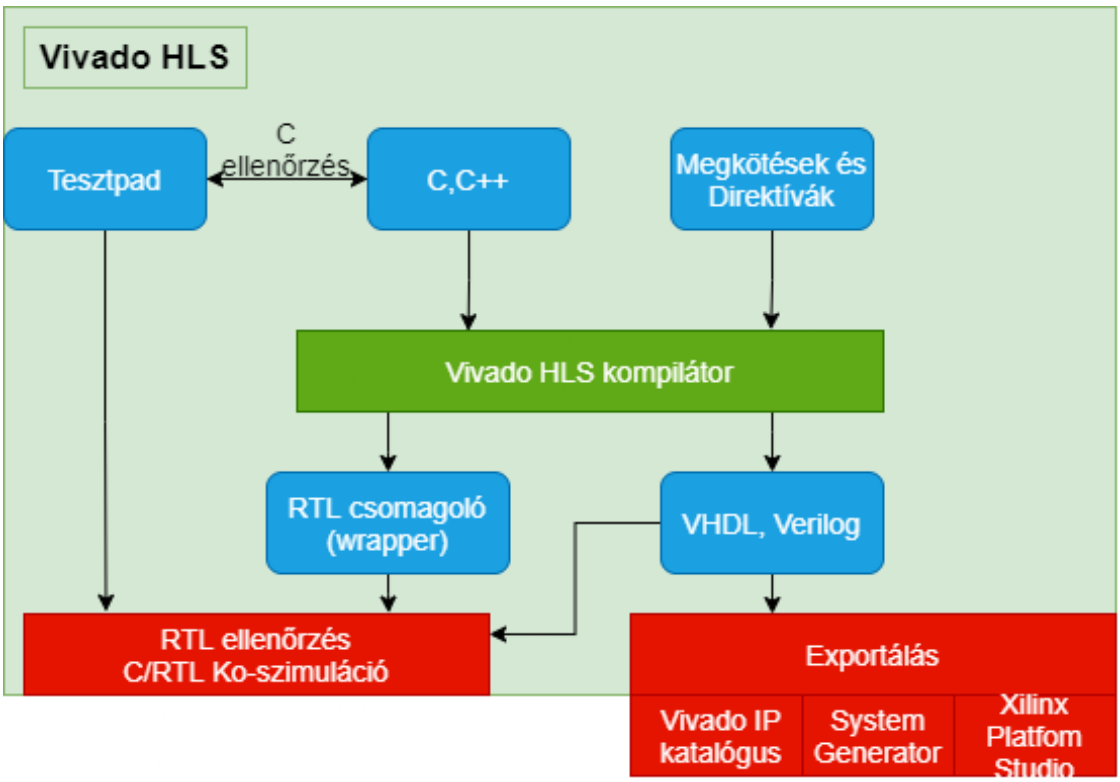
Miután a HLS eszköz elvégezte a kompilálást, kiosztotta az erőforrásokat, végrehajtotta az ütemezést és az összekapcsolást a következő és egyben utolsó lépés az RTL modell generálása az előzőleg meghozott tervezési döntések alapján.

Az erőforrás kiosztás ütemezés és összekapcsolás végezhető egyidőben is, de mivel mind összefüggenek egymással, a folyamat nagyon bonyolulttá válna, ezért általbában sorban végzik el őket a HLS eszközök.

Az RTL alapú architektúra regiszter szintű alkotó elemekből épül fel. Általában tartalmaz egy vezérlőt és egy adatútat. Az adatút tároló egységekből (regiszterek, memóriák stb.), funkcionális egységekből (aritmetikai logikai egységek, szorzó és eltoló áramkörök stb.) és a kapcsolatot biztosító elemekből (multiplexerek, sínek) áll. Az adat be- és kimenetek az adatúthoz vannak kapcsolva míg a kontroll jelek a vezérlőhöz. A kontroller egy véges állapotú gép (FSM), amely a kontroll jelek segítségével összehangolja az adatok áramlását az adatúton [4].

2.1.6. Vivado HLS

A Vivado HLS a Xilinx HLS eszköze, ami C, C++ vagy SystemC leíró nyelvekből generál VHDL, Verilog és SystemC RTL leírásokat. A generált leírást IP magba lehet csomagolni. Az IP magok beilleszthetőek a Vivado tervezői felületébe [5].



1. ábra: HLS alapú hardver kialakításának folyamata [6]

A funkcionális leírásban található függvényeket a HLS külön RTL modulként értelmezi, melyeknek a paraméter listán keresztül lehet meghatározni a ki és bemeneteiket. A függvény paraméterei átadhatóak érték szerint (adat port), cím szerint mutatóval (olvasható és írható be- és kimeneti porton), cím szerint tömbbel (alapértelmezetten memóriaként van kezelve, írható és olvasható). A függvénynek lehet visszatérítési értéke, ami az ap_return nevű kimeneti portot generálja. Minden RTL modul rendelkezik egy bemeneti órajel és reset porttal, opcionálisan

beállítható egy órajel engedélyező port is. A HLS eszköz vezérlőjeleket rendel az RTL modulokhoz. Az ap_ctrl_hs kényszerfeltétel beállításával a modulhoz generálódik az ap_start jel, amivel lehet engedélyezni a modul működését, az ap_idle jelzi a modul állapotát, az ap_done jelez, ha a modul befejezte a feladatát, az ap_ready jelez, ha a modul készen áll a következő adat fogadására. Az ap_ctrl_chain opciót beállítva az előbb felsorolt jelek mellé generál egy ap_continue jelet is, amellyel lehet jelezni a modul számára, hogy folytassa a művelet végzést. Az s_axilite AXI sínrendszerre illeszthető interfészt generál. Az ap_ctrl_none protokollt kiválasztva nem generálódnak modul szintű vezérlőjelek. A modul szintű protokollok mellett léteznek függvény argumentumokhoz rendelt protokollok is. Ilyenek az ap_none, ap_stable, ap_ack, ap_vld, ap_ovld és ap_hs. Ha ezen direktívák valamelyike be van állítva egy függvény argumentumhoz, a HLS eszköz hozzáadja az illető jelet a függvényből létrehozott modulhoz.

Fontosabb kényszerfeltételek [7]:

- Allocation: Korlátozza az erőforrás kiosztást, több modul fogja ugyanazt az erőforrást felhasználni
- Array_map: Több kisebb tömböt egy tömbbe egyesít, csökkentve a felhasznált BRAM-ok számát.
- Array_partition: Tömbök helyett regisztereket használ, ezáltal csökkentve az adatokhoz való hozzáférési időt.
- Array_reshape: Csökkenti a tömbök elemszámát azáltal, hogy megnöveli a szóméretet, javítva a hozzáférést anélkül, hogy több BRAM-ra lenne szükség.
- Data_pack: Egy struktúra adatmezőit egy nagyobb szószélességű skalárba helyezi
- Dataflow: Feladatszintű csővezetékesítés, növeli az RTL modul egyidejűségét és az áteresztőképességet
- Dependence: Információt szolgáltat, ami segíthet kiküszöbölni az iterációk közötti függőségeket. A HLS automatikusan azonosítja a ciklusokban a függőségeket, azonban bizonyos esetekben hamis függőségeket detektálhat, amik megakadályozzák a ciklusok csővezetékesítését. A dependence direktíva által jelezhetjük a HLS-nek, hogy valóban létezik-e függőség vagy nem.
- Expression_balance: A C alapú programokban megjelenhetnek olyan kifejezések melyek több művelet láncolatából épülnek fel az RTL modulban. Kis órajel esetében ez növelheti a rendszer késleltetését. Alapértelmezetten a HLS átrendezi a műveleteket az kommutativitás és asszociativitás tulajdonságoknak megfelelően, hogy egy

kiegyensúlyozott fát hozzon létre, csökkentve a késleltetést, többlet hardver felhasználás által. Az expression_balance direktívával ezt a funkciót kilehet kapcsolni.

- Function_instantiate: Minden függvény példányosítás esetén egy egyedi RTL megvalósítás keletkezik. Ez lehetővé teszi, hogy a minden példány helyileg legyen optimalizálva a függvény argumentumainak megfelelően.
- Inline: Eltünteti az adott függvényt a függvényhívás helyére való behelyettesítés által. A függvény ezután nem lesz egy külön álló egység RTL szinten. Néhány esetben ez lehetővé teszi a hatékonyabb optimalizálást a függvény körüli műveletekkel
- Interface: Meghatározza hogyan lesznek az RTL modul portjai generálva az interfész szintetizálása közben
- Latency: Megadható egy maximum vagy egy minimum késleltetési érték, vagy mindkettő.
- Loop_flatten: Lehetővé teszi egymásba ágyazott ciklusok egy cikluskénti való kezelését, csökkentve a szükséges órajelciklusok számát és néhány esetben jobb optimalizációs lehetőségeket kínálva
- Loop_merge: Egymás után következő ciklusokat egy ciklusba von össze csökkentve a késleltetést, növelve az erőforrás használatot, javítva a logikai optimalizációt.
- Loop_tripcount: Manuálisan beállítható a ciklus által elvégzendő ismétlések száma. Nincs hatással a szintézisre, csak analízisre használható
- Pipeline: Csővezetéket hoz létre, engedélyezve a műveletek párhuzamos végrehajtását, csökkentve az inicializálási intervallumot.
- Protocol: Meghatároz egy kód részt, melyben nem történnek órajel műveletek. Ez a rész használható egy interfész megadására, ami biztosítja, hogy a végső rendszer összekapcsolható lesz más hardverekkel ugyanazzal az interfész protokollal.
- Reset: Reset jelet ad hozzá globális vagy statikus változókhoz.
- Resource: Meghatározza, hogy egy tömb, aritmetikai művelet, vagy függvényargumentum milyen könyvtári erőforrással legyen megvalósítva.
- Stream: A tömb megvalósítására FIFO-t használ BRAM helyett. Ez hatékonyabb megoldás adatfolyamok esetében.
- Unroll: Kiterjeszti a ciklusokat, lehetővé téve azok iterációinak párhuzamos végrehajtását. Az RTL leírásban a ciklus tartalma többszöröződik.

A rendszerben felhasznált kényszerfeltételek a későbbiekben részletesen, példákon keresztül lesznek bemutatva.

2.1.7. HLS előnyei RTL szintű tervezéshez képest

- Csökkenti a tervezési időt. A tervező a rendszer viselkedésének leírására kell koncentráljon, a mikroarchitektúra részletes megvalósítása helyett. Megvalósítás során kisebb a valószínűsége hibák generálásának a fejlesztő által.
- Az ellenőrzési folyamat gyors. A rendszer viselkedését szoftveres eszközökkel tesztelhetjük egyszerűbben és gyorsabban.
- A tervezői lehetőségek feltárása (DSE) gyorsabb. A mikroarchitektúra változtatható a HLS eszköznek megadott kényszerfeltételek által, a forráskód változtatása nélkül. RTL szintű fejlesztésnél ez csak a forráskód jelentős részének megváltoztatásával lehetséges.
- A célplatform változtatható. A HLS képes a kiválasztott platformnak megfelelően újraütemezni a műveleteket.
- A HLS eszköz szoftver fejlesztők által is használható. A VHDL vagy Verilog nyelvekben történő fejlesztés alapos hardver ismereteket igényel. A HLS figyel a hardveres tervezés részleteire, megkönnyítve a tervező munkáját.
- A tervezési és ellenőrzési idő lecsökkentése által csökkenek a fejlesztési költségek és a piacra kerülési idő [8].

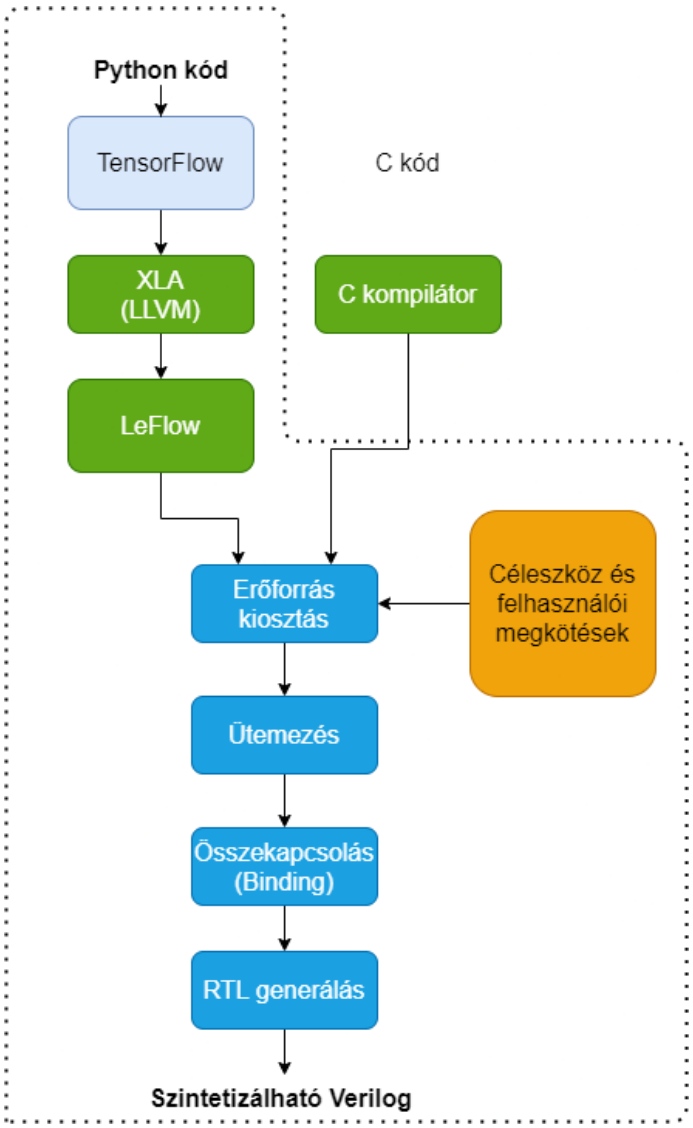
2.1.8. HLS hátrányai RTL szintű tervezéshez képest

- Jelenleg még a legújabb HLS eszközök sem biztosítanak annyira jó teljesítményt és erőforrás felhasználást, mint a kézzel írt RTL modellek, de sok esetben megközelíthetik azt [8].

2.2. Hasonló keretrendszer - LeFlow

A LeFlow keretrendszer 2018. július 14.-én jelent meg. A TensorFlow által meghatározott specifikációt fordítja le LLVM kompatibilis (Low Level Virtual Machine) specifikációra a Google XLA (Accelerated Linear Algebra) kompilátorával. Az XLA kompilátor a tensorFlow által generált gráfot, egy sor számítási maggá alakítja. Ezek a számítási magok egyediek lesznek az adott modellre, kihasználják az illető modellben rejlő optimalizálási lehetőségeket, ezért optimálsabbak a TensorFlow modellnél. Az LLVM kódból a LegUp magas szintű szintetizáló eszköz generál RTL leírást [10]. A felhasználó Pythonban, a TensorFlow keretrendszert használva, kell megvalósítsa a neuronhálót, emellett lehetősége van be beállítani különböző direktívákat, amelyekkel a ciklusok kiterjesztését, csővezetékesítését vagy a tömbök particionálását tudja szabályozni. A LeFlow tartalmaz automatizált tesztelést, ami a hardver modell eredményeit

összehasonlítja a python modell eredményeivel. Hátrányai közé tartozik, hogy olyan kerneleket használ, amelyek XLA-ban vannak megvalósítva és eredetileg CPU-khoz voltak tervezve. Sokat lehetne javítani a LeFlow-al elért eredményeken, ha léteznének az XLA-ban direkt FPGA-hoz megvalósított kernelek. Másik gyengesége a LeFlownak, hogy nem tartalmaz egy automatizált memória particionálásra tervezett algoritmus. Neuronhálók esetében a nagyszámú súly és bemeneti paraméterek a memóriák olvasása szűk keresztmetszetet jelenthet a rendszer számára párhuzamosítás esetén. További hátránya, hogy nem támogat fixpontos számábrázolást, amivel kisebb erőforrásfelhasználást lehetne elérni [11]. A LeFlow fejlesztői arra számítottak, hogy más kutatók az LeFlow rendszer alá fognak kiépíteni saját HLS megoldásokat.



2. ábra: LeFlow keretrendszer folyamatábrája.

3. A rendszer specifikációi és architektúrája

3.1. Felhasználói követelmények

- A felhasználónak figyelnie kell a neuronháló méretének és dírektiva konfigurációjának meghatározásánál, hogy a generált hardver erőforrás szükséglete ne haladja meg a céleszközön található erőforrásokat.
- A bemeneteket az adott fixpontos számábrázolásnak megfelelően integer értékként kell átadni. A kimeneten kapott integer értékeket visszaalakítva az adott fixpontos számábrázolásnak megfelelően kell értelmezni. Például, ha 16 bites fixpontos számokat használ a rendszer, ahol 11 bit a törtrész, 5 bit az egészrész 2^11 – el kell felszorozni a bemeneti értékeket és 2^11-el kell visszaosztani a kimeneti értékeket. Így az 1- bemeneti érték helyett 2048-at kellene a bemenetre írni és a kimeneten kapott 2048- as érték 1-nek felelne meg.
- A neuronháló előre legyen tanítva és a betanított súlyzókat a header fájlban kell átadni

3.2. Funkcionális követelmények

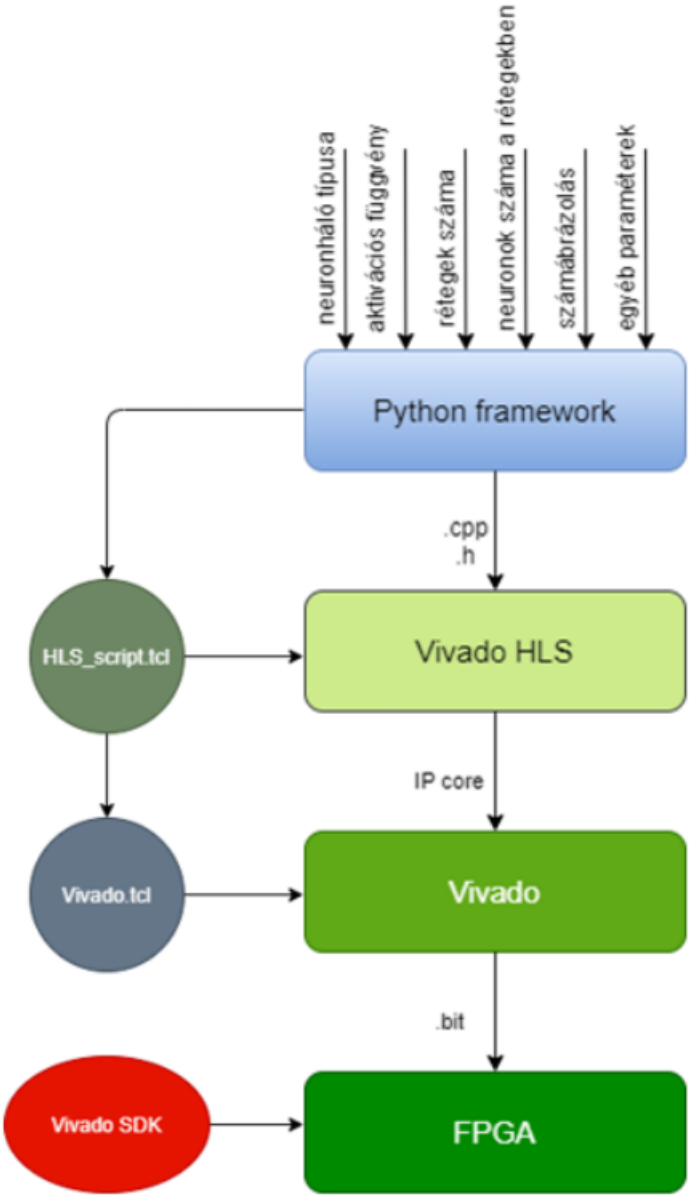
- A kigenerált hardver órajel periódusa ne legyen nagyobb a céleszköz maximális frekvenciájánál, hogy az a lehető legnagyobb sebességgel tudjon működni
- RBF és MLP típusú neuronhálók támogatása
- Szigmoid és Gauss aktivációs függvények
- Fixpontos számábrázolás 32 vagy 16 bites vagy általánosan n-bites adatokkal
- A neuronhálók konstansokon és makrókon keresztül legyenek paraméterezhetőek
- A neuronhálóhoz legyen megalósítva legalább egy interfész a bemeneti adatok beolvasására és a kimeneti adatok kiolvasására
- A rendszer tcl szkripttel legyen teljesen automatizálva

3.3. Nem Funkcionális követelmények

- Python 3 vagy annál újabb verziók python 3.9-ig
- Vivado HLS
- Vivado
- ARM processzorral rendelkező SoC (a rendszert a Zynq board xc7z010clg400-1 tokozású lapon teszteltük)
- Vivado HLS Command Prompt

- Nem operációs rendszer függő, Windows 10, Linux vagy MAC operációs rendszer egyaránt működtethető

3.4. A rendszer architektúrája

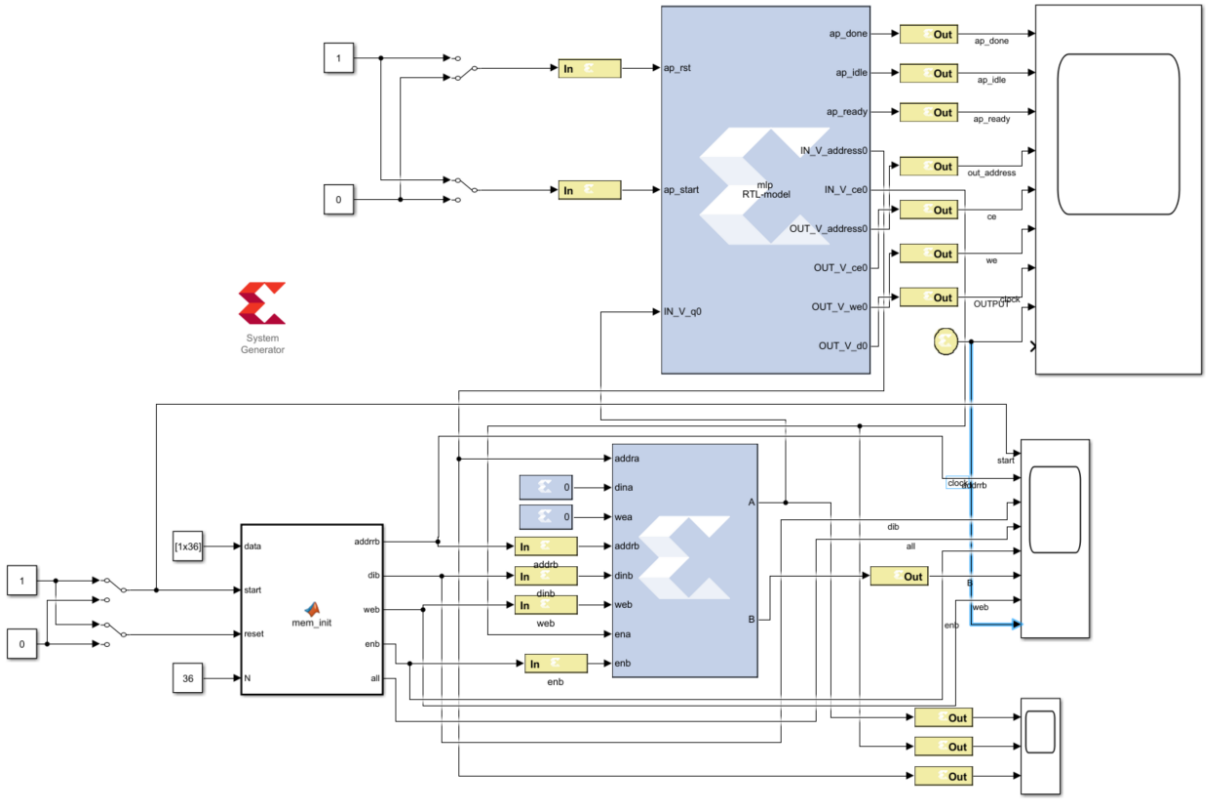


3. ábra: Rendszer folyamatábra.

A felhasználó Pythonban felépíti és paraméterezi a neuronhálót a keretrendszer által biztosított függvényeket használva. A keretrendszer elvégzi a tanítást, majd generálja a C++ állományt a Vivado HLS számára. A HLS eszköz szintetizálja a C++ kódot és exportálja a kiépített neuronháló IP magját. Az IP mag beillesztődik egy Vivado-ban felépített rendszerbe. A Vivado szintetizálja a rendszert, elvégzi az implementálást (fordítás, leképezés, elhelyezés, huzalozás) és generálja a bit

fájlt, majd felprogramozza az FPGA áramkört. Jelenleg egy interfész van megvalósítva a rendszerhez. A neuronháló bemeneteit az SoC-n levő Zynq processzoron keresztül lehet betölteni egy BRAM-ba, ahonnan majd a neuronháló kiolvassa azokat. A HLS és Vivado feladatainak automatizálása Tcl szkriptekkel van megoldva, amiket a Python keretrendszer indít el.

A rendszer ki van építve Simulink környezetben is System Generator használatával. Ennek nagy előnye, hogy megoldja az interfész kérdését. Egy Matlab modul segítségével számítógépről tudjuk küldeni a neuronháló bemeneti adatait USB-n keresztül.

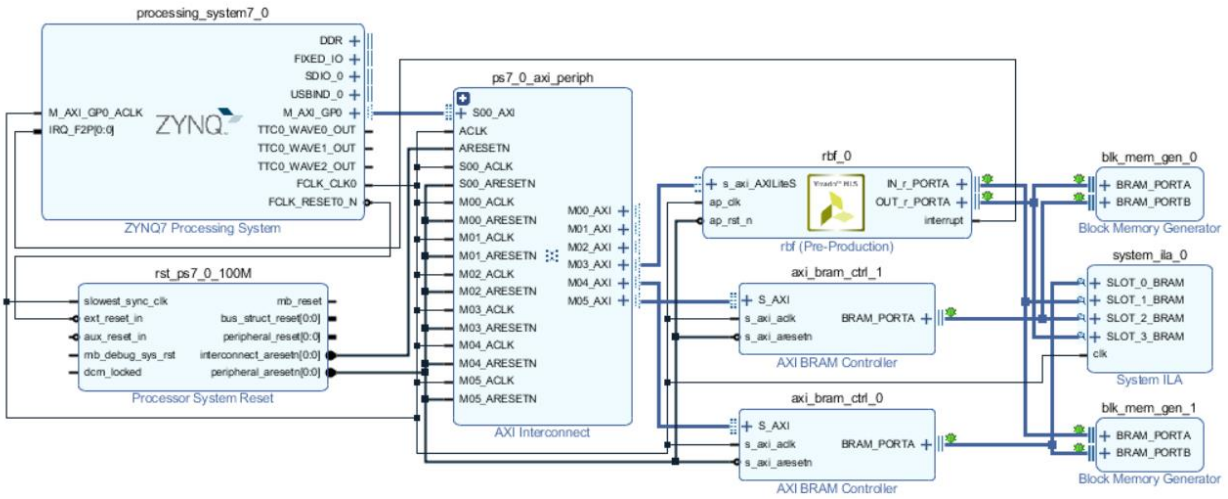


4. ábra: Simulink-ben System Generator-al felépített rendszer.

A felül látható nagy blokk a neuronháló, alatta található egy Dual Portos Block Ram, ebből olvassa ki a neuronháló a bemeneteket. Ettől balra található egy matlab modul, ami feltölti a memóriát a bemeneti adatokkal.

A szimulációkat a Simulink környezetben felépített rendszerrel végeztem.

A végső rendszer azonban Vivado-ban valósítottuk meg. A Vivado lehetővé teszi több különböző interfész kiépítését, például hálózati, USB vagy HDMI interfészeket.



5. ábra: Vivado-ban felépített hardver tömbvázlata.

A Zynq processzor a neuronháló bemeneteit AXI sínrendszeren keresztül beírja egy BRAM-ba a neuronháló ezeket az értékeket kiolvassa, majd kiszámolja a kimenetet. A kimeneti értékeket egy másik BRAM-ba írja, ahonnan a processzor egy axi sínrendszeren keresztül kiolvashatja azokat.

4. Megvalósítás

4.1. Módszerek

A neuronhálók megvalósítását a legkisebb funkcionális egységek implementálásával kezdem, majd ezeket összeillesztve építek fel bonyolultabb szerkezeteket. A kiindulási pont egy neuron megvalósítása, egy egyszerű lépcső aktivációs függvénnyel. Következő lépés, több neuron generálása által egy neuron réteg kialakítása. A rétegek többszörösítésével és azok összekapcsolásával egy többrétegű neuronháló megvalósítása. Ezután az alapegységek különböző topológiákba történő rendezése által, vagy más architektúrák megvalósításával különböző típusú neuronhálókat alakítok ki. A rendszer a későbbiekben bővíthető más aktivációs függvényekkel is. Az egységek tervezésénél lehetőséget biztosítok a paraméterezhetőségre és más egységekkel történő összekapcsolhatóságra. A mikroarchitektúra kényszerfeltételekkel történő optimalizálását funkcionális egységenként végzem. A legjobb mikroarchitektúrák feltárására érdekében Design Space Exploration (DSE) módszert alkalmazok, vagyis kipróbálok többféle direktíva konfigurációt és többféle megvalósítást egy adott modulra. Az eredményeket három fő szempont alapján értékelem ki: teljesítmény, erőforrás felhasználás mértéke és pontosság.

4.2. Többrétegű perceptron típusú háló

4.2.1. Az inger kiszámolása

Adott neuron ingerének a kiszámítása felfogható egy vektor szorzásként. A neuron bemeneteinek vektorát kell összeszorozni a neuronhoz tartozó súlytényezővektorral. Ez a művelet intenzív szorzást és összeadást igényel. Mivel ez a művelet igen gyakran előfordul FPGA alkalmazásoknál a gyártók speciális hardvert építettek az FPGA lapokba. DSP áramkörön megvalósítani a MAC (szorzás és összeadás) műveletet hatékonyabb, mint a programozható logikával. A modern FPGA-k több szász DSP modult tartalmaznak.

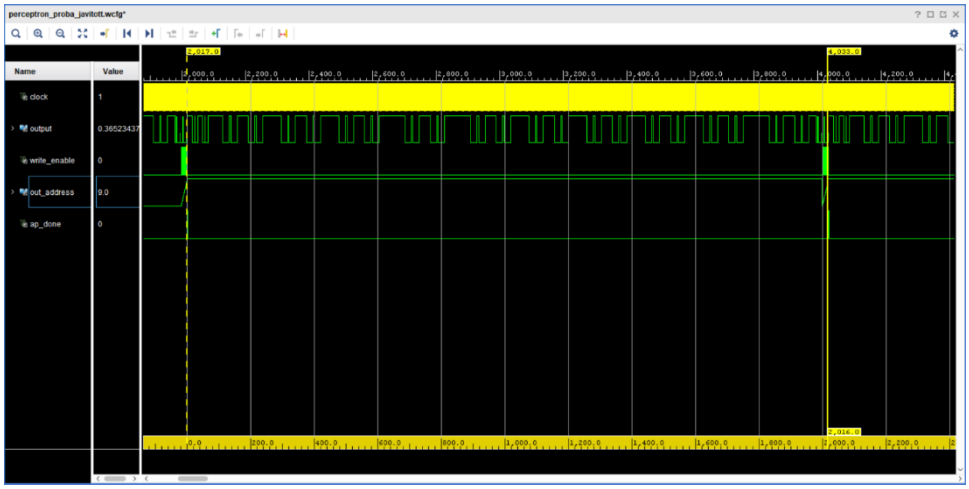
```
void vector_multiply(const MY_FIXED A[],
        const MY_FIXED B[], MY_FIXED * __restrict__ result,
        const unsigned size)
{
    MY_FIXED sum=(MY_FIXED)0;
multiply_loop:for(int k=0; k<size; ++k)
    {
#pragma HLS pipeline II=1
#pragma HLS unroll factor=4

        sum+=A[k]*B[k];
    }
    *result=sum;
}
```

6. ábra: Vektor szorzás direktívákkal.

A HLS minden függvényt külön modulként kezel. A függvény paraméterei jelentik a modul ki- és bemeneteit. Jelen esetben az A és B tömbök, valamint a size változó lesznek a modul bemenetei és a result pointer lesz a kimenet. Az A és B tömbök fogják tartalmazni a neuron bemeneteit és a súlyzóit, a size változóban kapja meg a modul a neuron bemeneteinek a számát. A tömbök elemei páronként szorzódnak össze, majd az eredmény hozzáadódik a sum változóhoz.

A direktívák hatásának szemléltetésére végzett bemutató méréseket egy 2 rétegű perceptron típusú neuronhálón végeztem. A neuronháló 1. rétegje 36, a 2. 10 neuront tartalmaz. Minden neuron szigmoid aktivációs függvényt használt. Az adatok tárolása és a műveletek elvégzése 24 bites fixpontos számokkal történt, ahol 15 bit képviselte az egész részt. A neuronháló által megoldott feladatot a későbbiekben részletezem.

Az alábbi mérésnél a mérőpálcákat az ap_done jel két egymásutáni 1-es értékére állítottam. A két mérés között 2000 mintavétel (órajel) van. Ez egy 100 MHz-es órajel ciklussal 20 mikrószekundumot jelent. Ennél a mérésnél nem használtam direktívákat.



7. ábra: Mérés direktívák használata nélkül.

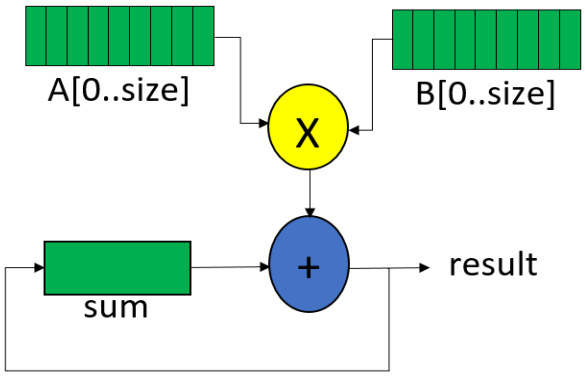
Ennek megfelelően a generált áramkör terület (erőforrás) takarékos. Az összes MAC művelet esetében ugyanaz a DSP van használva, így összesen csak egy DSP-re van szükség.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	63
FIFO	-	-	-	-
Instance	5	1	285	704
Memory	2	-	0	0
Multiplexer	-	-	-	376
Register	-	-	144	-
Total	7	1	429	1143
Available	120	80	35200	17600
Utilization (%)	5	1	1	6

8. ábra: Erőforrás felhasználás statisztika, direktívák használata nélkül.



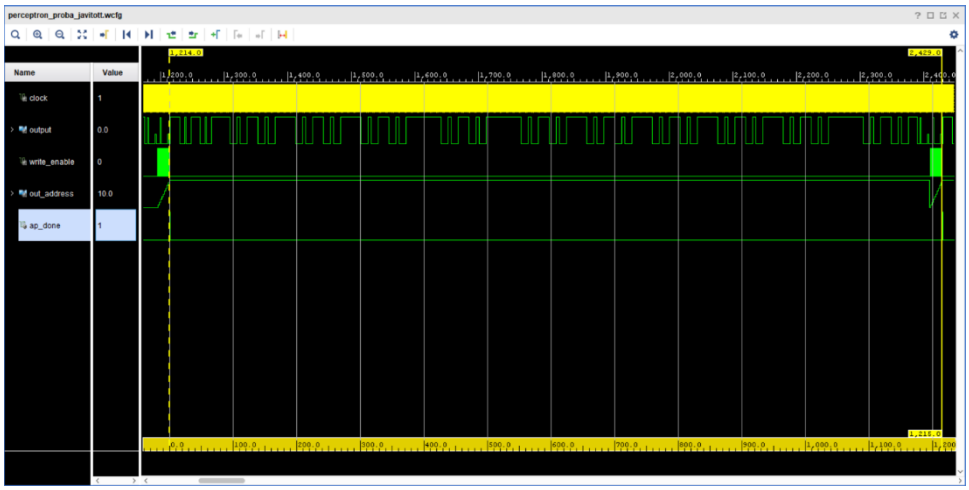
9. ábra: Vektor szorzás mikróarchitektúra direktívák alkalmazása nélkül.

A ciklusok kiterjesztése (Unrolling loops)

Alapértelmezetten a HLS a for ciklusokat szekvenciálisra szintetizálja. Ez egy erőforrás (terület) hatékony megoldás, mert ugyanazon a hardver részen végzi el sorban a műveleteket. A #pragma HLS unroll direktívát használva a ciklusok kiterjeszthetőek. A cikluson belül található műveletek sokszorosítva lesznek egy faktornak nevezett érték-szer. Amennyiben ciklus iterációinak száma nem a kiterjesztési faktor egész számú többszöröse, a HLS még egy for ciklusban elvégzi a fennmaradó iterációkat. Ha ezek az iterációk nem fontosok a feladat szempontjából, kihagyhatjuk őket a skip_exit_check argumentum megadásával. A legjobb esetben a ciklus iterációi nem tartalmaznak adatfüggőséget (egyik állítás sem függ egy előző iterációban generált adattól), ekkor a ciklus teljesen kiterjeszthető. Ha nem adjuk meg a faktort akkor a HLS teljesen kiterjeszti a ciklust, ami jó megoldás lehet kisebb ciklusok esetében, de nagy iteráció számnál, mint például 1 millió a teljes kiterjesztés megvalósíthatatlan a hardver korlátok miatt [12].

Jelen esetben a ciklusban 4 művelet van: olvasás az A tömbből, olvasás a B tömbből szorzás és az összeadás. Az olvasás és szorzás műveletek végig függetlenek egymástól. Az összeadási művelet függ az akkumulátor előző értékétől, olvasás írás után adatfüggőség lép fel. A ciklust 4-es faktorral terjesztem ki, hogy elkerüljem az adatfüggőséget.

Az alábbi mérésnél a ciklus 4-es faktorral volt kiterjesztve. Egyszerre 4 MAC műveletet végez el, ennek megfelelően 4 DSP-t is használ. A művelet elvégezhető 1215 órajel alatt.



10. ábra: Mérés, vektor szorzás kiterjesztve 4-es faktorral.

Utilization Estimates

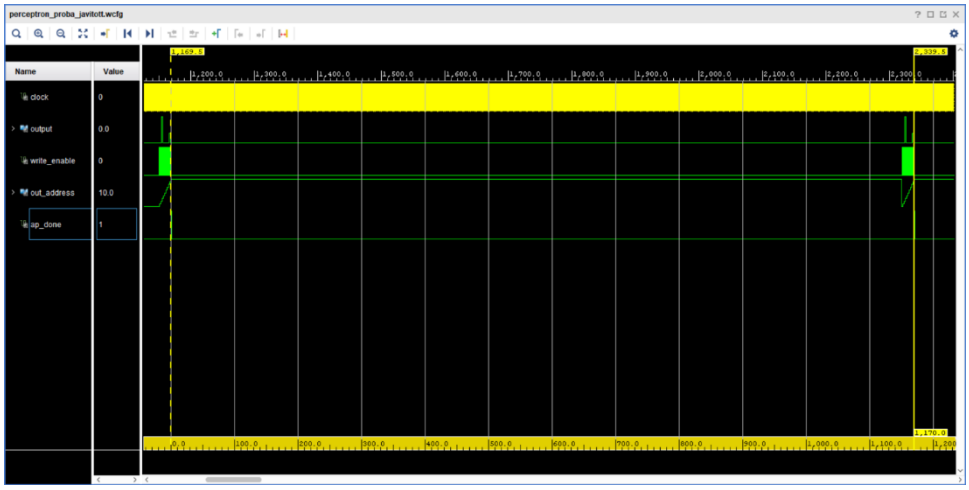
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	98
FIFO	-	-	-	-
Instance	5	4	620	1844
Memory	0	-	68	20
Multiplexer	-	-	-	324
Register	-	-	45	-
Total	5	4	733	2286
Available	120	80	35200	17600
Utilization (%)	4	5	2	12

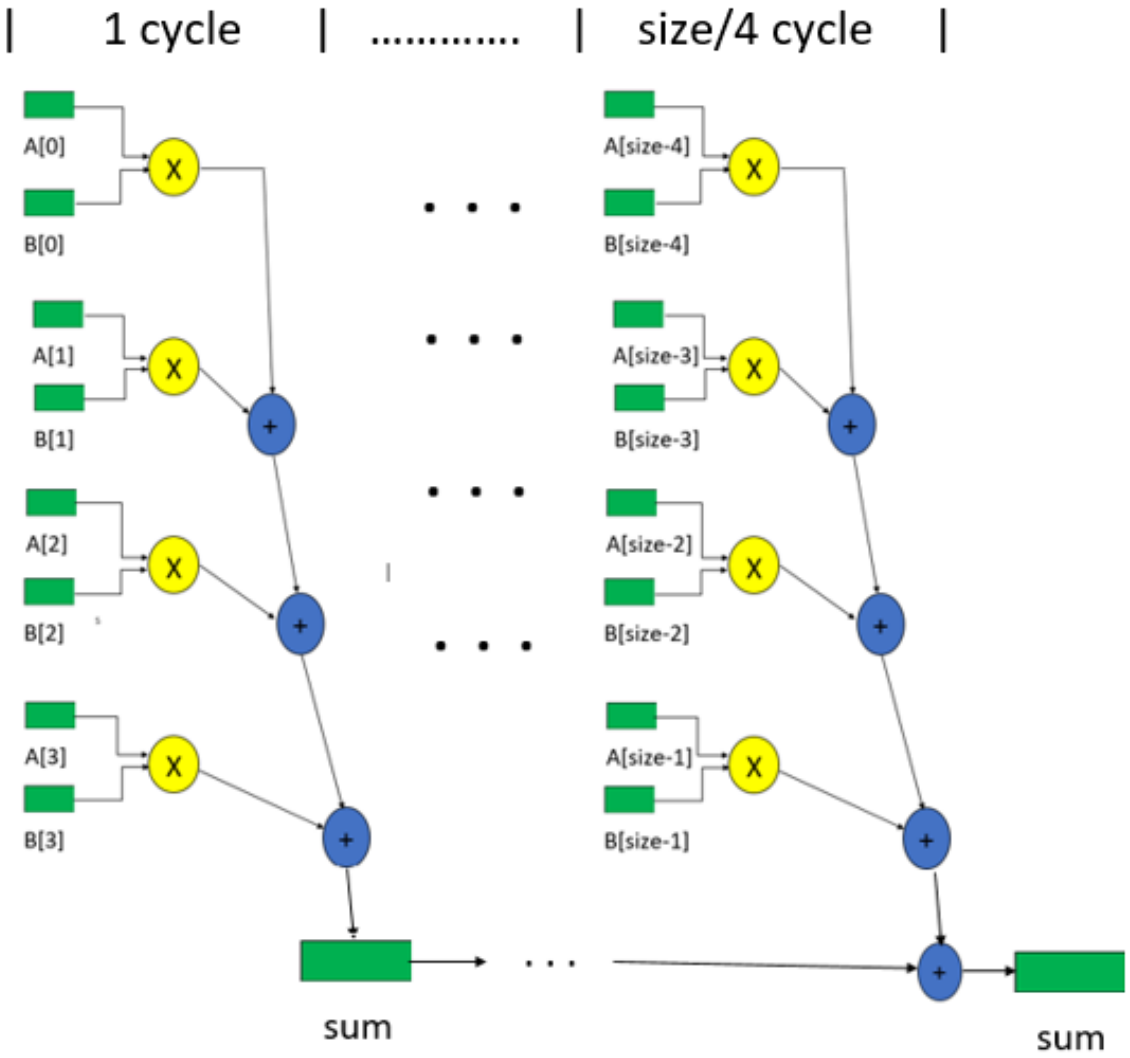
11. ábra: Erőforrás felhasználási statisztika, vektor szorzás kiterjesztve 4-es faktorral.

A ciklusok kiterjesztése által a teljesítmény nagyban növelhető feltéve, ha figyelünk az írás és olvasás műveletekre. A tömbök értékei általában BRAM-okban vannak eltárolva. Több típusú BRAM létezik, az FPGA lapokban leggyakrabban használt BRAM-oknak két port-ja van, amelyeket lehet használni írásra vagy olvasásra, így 1-nél több olvasási és írási művelet nem végezhető el párhuzamosan. Erre a problémára megoldást jelent, ha tömb értékeit külön regiszterekbe tároljuk, mert egy regiszter írható és olvasható is egy órajel alatt. A HLS-t a #pragma HLS array_partition variable=_tömb_neve_ complete direktívával lehet utasítani tömb értékeinek külön regiszterekbe való eltárolására. Jelen esetben a 4-es kiterjesztésnek köszönhetően 4 olvasást kell elvégezzen a rendszer mindkét tömbből egyszerre.

Az A és B tömbök partícionálása által a rendszer számítási ideje tovább csökken 1170 órajelre.



12. ábra: Mérés, tömbök particionálva.

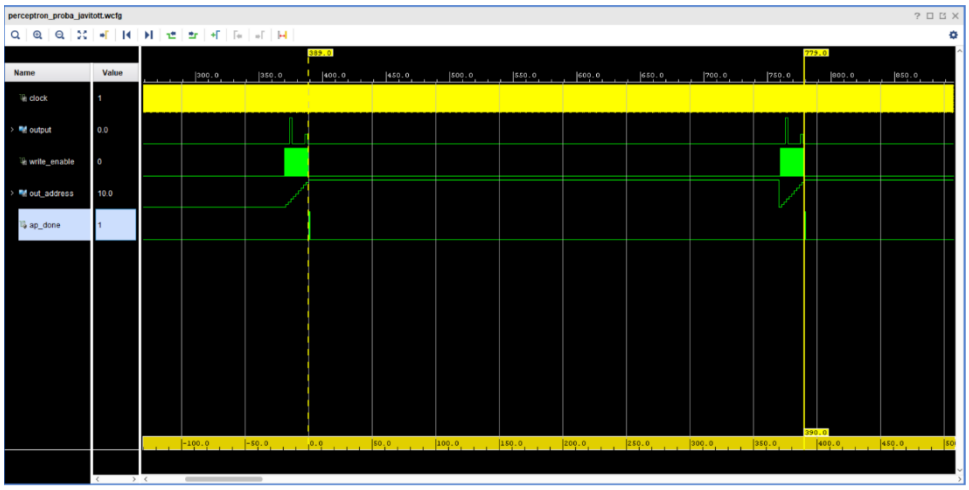


13. ábra: Vektor szorzás mikróarchitektúra, ciklus kiterjesztve, tömbök partícionálva.

Ciklusok csővezetékezése (Loop pipelining)

A ciklusok kiterjesztésén kívül lehetőség van a ciklusok iterációinak konkurens, de késleltetett végrehajtására. Sok esetben a ciklusok nem kiterjeszthetőek, viszont a következő iteráció megkezdhető még az előző befejezte elött, vagyis az iterációk konkurensen hajthatók végre egy bizonyos késleltetéssel.

A mérés eredménye 390 mintára csökkent, a csővezeték alkalmazása után a ciklusban. A felhasznált erőforrások száma alig növekedett.



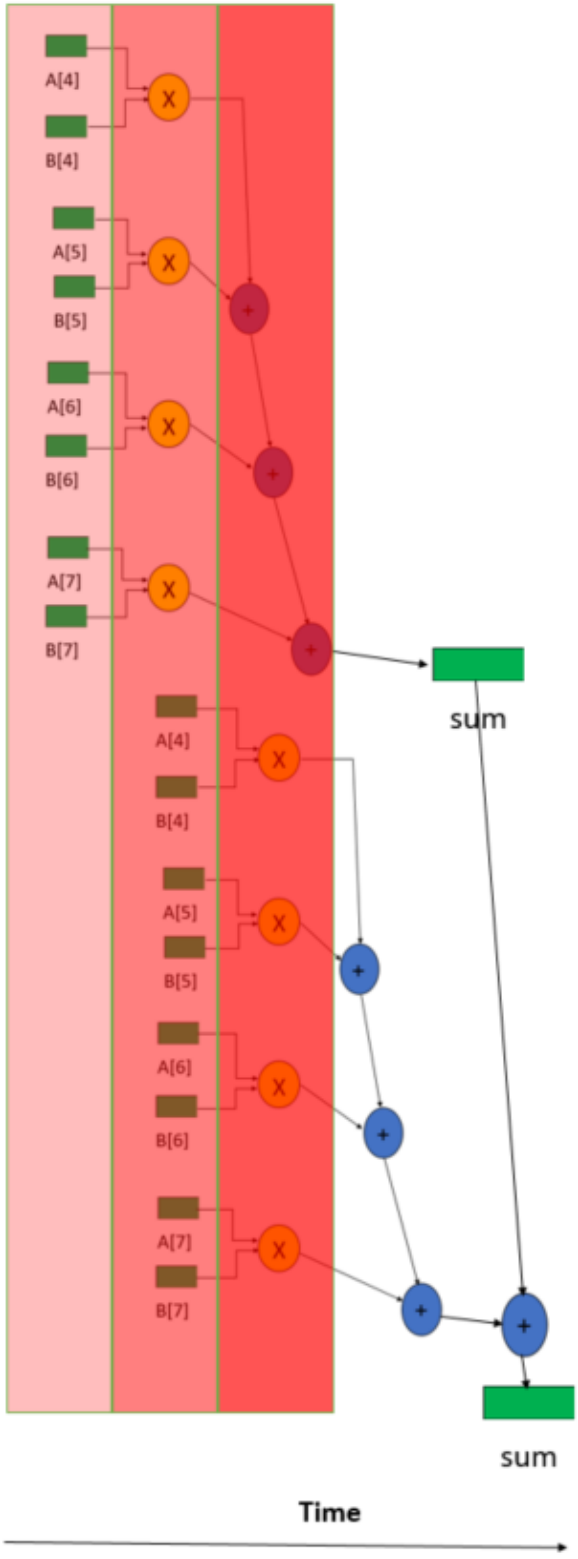
14. ábra: Mérés, vektor szorzás csővezetékezve.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	453
FIFO	-	-	-	-
Instance	3	4	2139	4013
Memory	0	-	20	6
Multiplexer	-	-	-	206
Register	-	-	1375	-
Total	3	4	3534	4678
Available	120	80	35200	17600
Utilization (%)	2	5	10	26

15. ábra: Erőforrás felhasználási statisztika, vektor szorzás csővezetékesítve.



16. ábra: Vektor szorzás, ciklus kiterjesztve, tömbök partícionálva, ciklus csővezetékesítve.

4.2.2. Egy réteg felépítése

Egy réteg az általa tartalmazott neuronokból épül fel. A réteg bemenete megegyezik neuronjainak a bemenetével. A réteg súlymátrixában soronként vannak eltárolva az egyes neuronok súlyzói.

```
void single_layer(const MY_FIXED IN[],
        const MY_FIXED W[][greatest_layer_neurons],
        unsigned neuron_number, MY_FIXED OUT[output_number])
{
    MY_FIXED y;

    neuron_loop:for(unsigned i=0;i<neuron_number;++i)
    {
#pragma HLS unroll
        single_neuron(IN, W[i], &y);
        OUT[i]=y;
    }
}
```

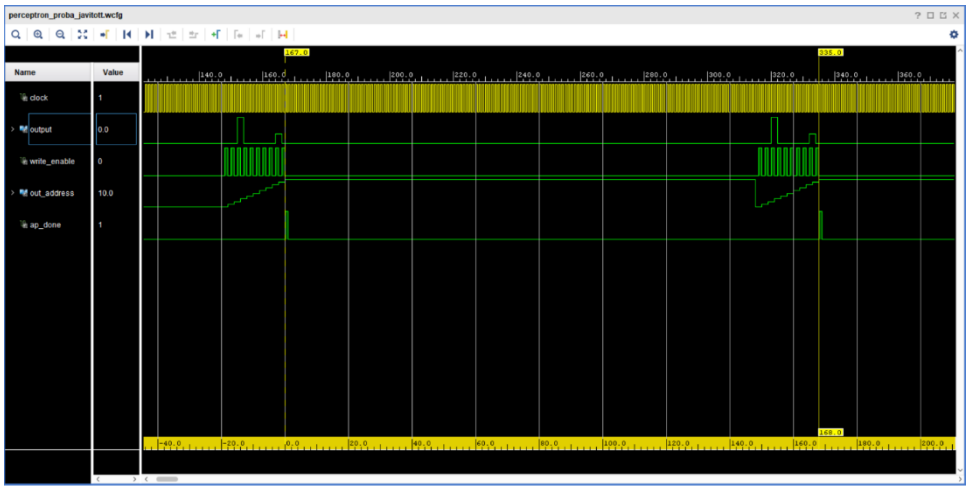
17. ábra: Egy neuron réteg.

Egy neuron szerepe a saját ingerének átvezetése az aktivációs függvényén. Ennél a feladatnál szigmoid aktivációs függvény van használva.

```
void single_neuron(const MY_FIXED X[perceptron_input],
        const MY_FIXED W[], MY_FIXED * __restrict__ y )
{
    //inger kiszámolása
     vector_multiply(X,W,y,perceptron_input);

    // aktivacios fuggveny

    //step_function(y);
     sigmoid(y);
}
```

18. ábra: Egy neuron.

Az eddigi mérések során a neuronok szekvenciálisan dolgoztak. A réteg neuronjain iteráló ciklust kiterjesztem. A következő mérésben a neuronok már párhuzamosan dolgoznak. A mérés értéke több mint felére csökken 168 mintára, de a felhasznált erőforrások száma nagy mértékben megnő, még egy ilyen relatív kis méretű neuronháló esetében is.



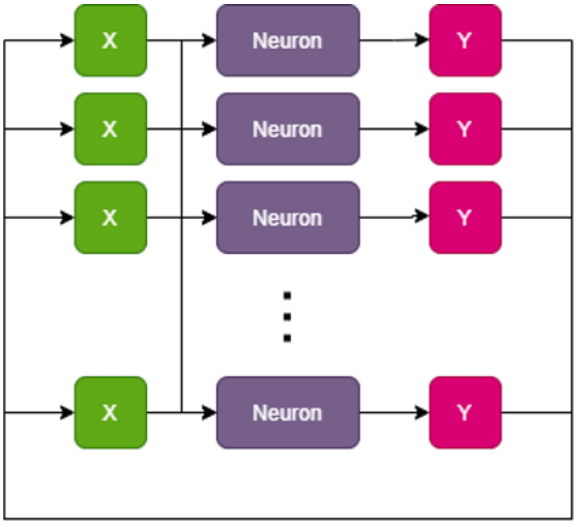
19. ábra: Mérés, neuronok párhuzamosítva.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	300
FIFO	-	-	-	-
Instance	70	40	8690	30700
Memory	-	-	-	-
Multiplexer	-	-	-	59
Register	-	-	981	-
Total	70	40	9671	31059
Available	120	80	35200	17600
Utilization (%)	58	50	27	176

20. ábra: Erőforrás felhasználási statisztika, neuronok párhuzamosítva.



21. ábra: Neuronok kiterjesztve (párhuzamosítva) mikróarchitektúra.

4.2.3. Szigmoid aktivációs függvény szakaszonkénti lineáris közelítéssel

A logisztikus vagy szigmoid karakterisztikájú aktivációs függvény lehetővé teszi olyan nem triviális problémák megoldását egy többrétegű perceptron típusú neuronháló számára, amelyeket csak egy nemlineáris aktivációs függvénnyel lehet megoldani.

$$f(x) = \frac{1}{1 + e^{-x}} \qquad \textbf{1.}$$

Szigmoid függvény képlete

Azonban egy nemlineáris függvény egyenes megvalósítása FPGA áramkörön nem lehetséges. Sokféle közelítési módszer létrejött az évek során. Ezek közé tartozik az exponenciális tag kiszámolása Taylor sorbafejtéssel, szakaszonkénti lineáris közelítés (piecewise linear approximation), keresőtáblázat (look up table). A Taylor sorbafejtés előnytelen, mert sok szorzásra van szükség, ezáltal nagy lesz az erőforrás szükséglet és a késleltetés. A keresőtáblázat a leggyorsabb módszer. Hátránya, hogy sok memóriára van szükség az értékek eltárolásához, ha nagy pontosságot szeretnénk elérni. Én a szakaszonkénti lineáris közelítést használtam, ami csak egy szorzással és egy összeadással igényel több műveletvégzést, mint a keresőtáblázat viszont lényegesen kevesebb értéket kell eltárolni.

$$f(x) = \begin{cases} k_0 x + b_0, & x < x_1 \\ k_1 x + b_1, & x_1 < x < x_2 \\ \dots & \\ k_n x + b_n, & x_n < x \end{cases} \qquad \textbf{2.}$$

Szakaszonkénti lineáris közelítés általános képlete

Figyelembe vettem, hogy a szigmoid aktivációs függvény szimetrikus, ezért elegendő, csak egyik felét meghatározni a szakaszonkénti lineáris közelítéssel. A másik fele számolható a f(-x) = 1 - f(x) képlet alapján. Az közelítési intervallumok számát és a hozzájuk tartozó k és b értékek a keretrendszer python része generálja. A k és b értékeket egy kétdimenziós tömbbe tárolom, amelyet direkt lehet címezni az x egész számnak megfelelő értékével. Ha az x nagyobb a legfelső intervallum felső határánál a szigmoid értéke 1 lesz. Ha kisebb a legalsó intervallum alsó határánál az érték 0 lesz.

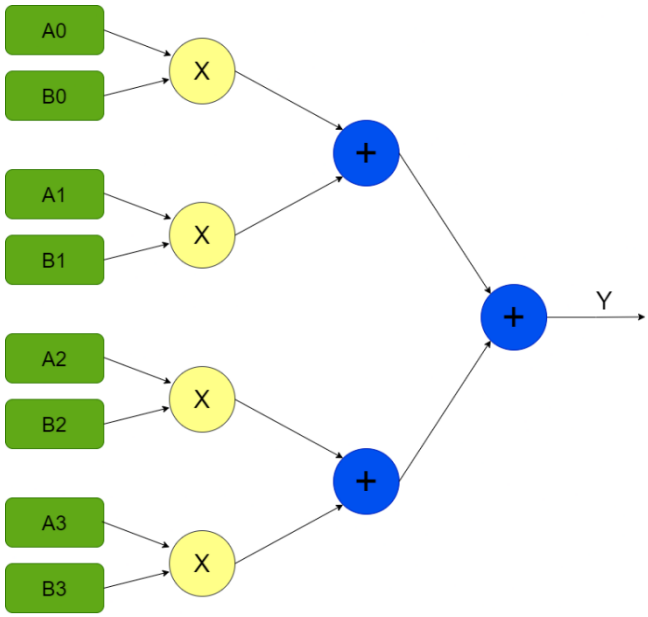
4.2.4. Szigmoid aktivációs függvény keresőtáblázattal

Ebben az esetben is figyelembe vettem, hogy a szigmoid függvény szimmetrikus így [0,5) intervallumra generáltattam az értékeket. A (-5,0) intervallum értékeit kitudom számolni a f(-x) =

1 – f(x) képlet alapján. Az intervallumon 2 a -7. -en lépéssel vett számokra generáltattam a szigmoid függvény megfelelő értékeit, 641 elemből álló keresőtáblázatot kaptam. Ez a lépésméret elég nagy pontosságot biztosít, sok feladat esetében kisebb pontosság is megfelelő. Mivel ez a modell csak egész számokat használ ezért a szigmoid bemenetére kapott értéket háromszor jobbra eltolva, az érték egyenesen használható a keresőtáblázat címzésére. A keretrendszer lehetőséget fog biztosítani a felhasználó számára, hogy kiválassza a számára szükséges pontosságot és beállítsa a szigmoid aktivációs függvény meredekségi, valamint eltolási paramétereit. Ezek alapján fogja kigenerálni a megfelelő keresőtáblázatot.

4.2.5. Egész alapú számábrázolás használata

A generált mikroarchitektúra vizsgálata során rájöttem, hogy az ap_fixed könyvtárból használt fixpontos számokkal végzett műveleteknél sok erőforrás van használva a számok szaturálásának és kerekítésének megoldására. A többlet erőforrás felhasználás mellett az architektúra módosításának lehetőségeit is korlátozza. Fixpontos számokat használva kerekítéssel és szaturálással a HLS nem volt képes kigenerálni a legkevesebb ciklust igénylő fa architektúrát 22. ábra a vektor szorzás elvégzésére.



22. ábra: Fa típusú architektúra.

Megoldásként áttértem fixpontos számokként értelmezett integerek használatára. int16_t adattípust használtam a bemenetek, súlyvektorok és a szigmoid keresőtáblázat esetében, míg int64_t típust az inger kiszámolásánál keletkező adat tárolására. A 16 bites számokat, úgy értelmeztem, hogy 5 bit képviselte az egészrészt és 11 bit a törtrészt, így az egész számok fixpontos

számmá alakításához 2048-al kell leosztani, 2048 a skála. Két 16 bites integer szorzása során kapott eredmény eltárolására legtöbb 32 bitre van szükség. Két 32 bites érték összeadásánál kapott eredmény eltárolására legtöbb 33 bitre lehet szükség. Ezek alapján, a rendszerben biztosan nem lesz túlcsordulás, ha kevesebb mint 33 neuront használunk egy rétegben. Ha ennél többet szeretnénk használni, előfordulhat túlcsordulás.

Utilization Estimates

Summary

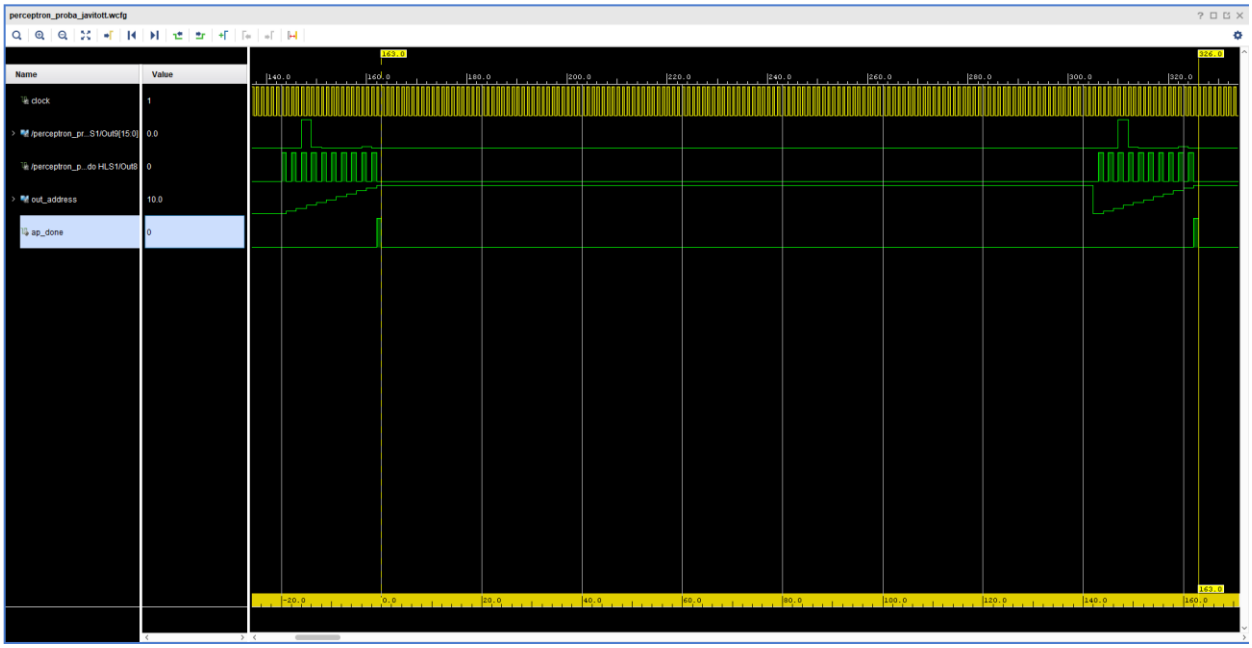
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	346
FIFO	-	-	-	-
Instance	5	35	3381	1640
Memory	-	-	-	-
Multiplexer	-	-	-	59
Register	-	-	758	-
Total	5	35	4139	2045
Available	120	80	35200	17600
Utilization (%)	4	43	11	11

Detail

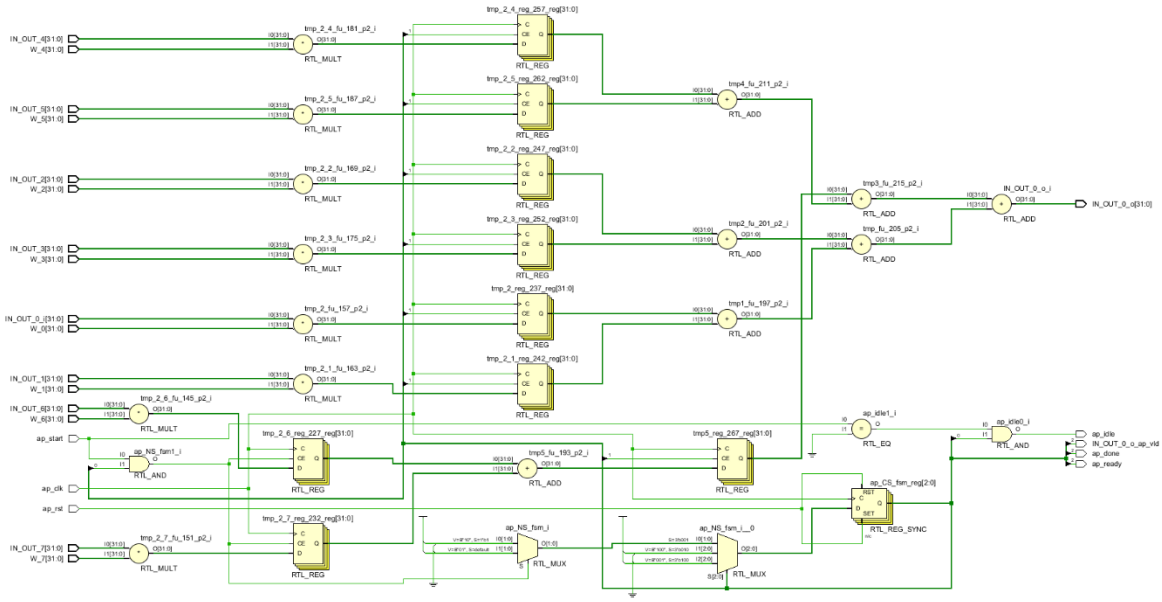
23. ábra: Erőforrás felhasználási statisztika, inger kiszámolása fa típusú architektúrával, neuronok csővezetékesítve.

Az integer számos rendszer optimális architektúrájának a vektor szorzás teljes kiterjesztését és a neuronok csővezetékesítését találtam. Ebben az esetben annyi DSP-re lesz szükség amennyi bemenete van a rendszernek, vagy a legnagyobb rétegben található neuronok száma, amennyiben ez az érték nagyobb a bemenetek számánál. Az így kapott rendszer késleltetése 163 µs lett 24. ábra, ami 5 µs-al gyorsabb, mint a fixpontos rendszer párhuzamosított neuronokkal és lényegesen kevesebb erőforrást használt fel 23. ábra.

Ha teljesen kiterjesztem a neuronokat rendszert a késleltetése tovább csökken 156 µs-ra , de a DSP modul felhasználás a késleltetés csökkenésének mértékével aránytalanul nagyra nő, mert ebben az esetben minden neuronnak szüksége lesz 35 DSP modulra, ez 10 neuron esetében 350 DSP modult jelent.



24. ábra: Mérés, integer műveletekkel, vektor szorzás kiterjesztve, neuronok csővezetékesítve



25. ábra: 6 elemre generált fa típusú vektorszorzás architektúra.

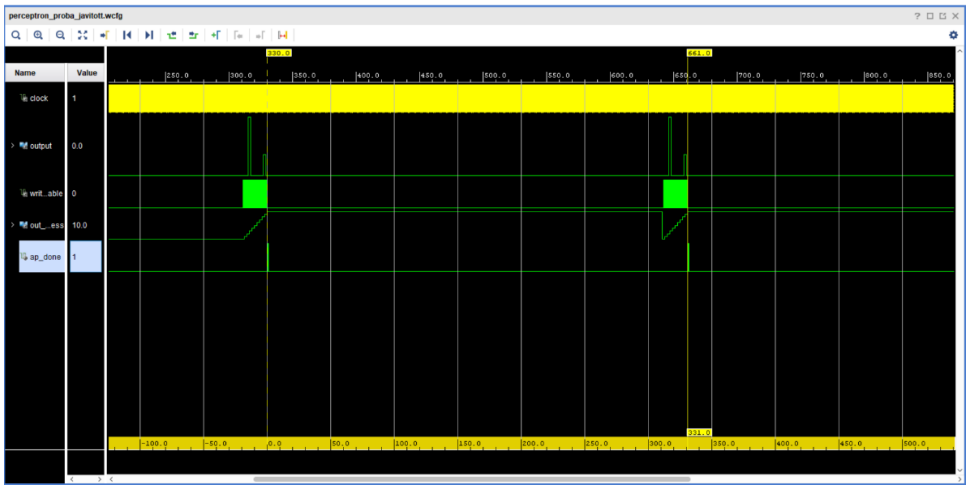
4.3. Fix pontos számábrázolás lebegőpontos számábrázolással szemben

A HLS-ben megvalósított neuronhálók megvalósításánál fixpontos adattípusokat használtunk. A fixpontos számábrázolásnak számos előnye van a lebegőpontos számábrázoláshoz képest. A fixpontos rendszerek mindig jóval kevesebb erőforrást használnak. Kevesebb digitális jelfeldolgozó processzor, kereső táblázat és bistabil áramkör szükséges fixpontos számokkal való munka során. Kisebb lesz az energia fogyasztás a kevesebb erőforrás felhasználása miatt. A

számítások gyorsabbak, így kisebb a késleltetés. A számítások pontossága megközelíti a lebegőpontos számításokét. Érdemes alkalmazni a fixpontos adattípust, ha a kis pontosság vesztés nem befolyásolja az alkalmazás helyességét [13].

A különböző adattípusokkal készített méréseket, az MLP HLS modell bemutatásánál használt neuronhálóval végzem, ugyanazzal a bemenettel, a legutolsó direktíva konfigurációt felhasználva. A lebegőpontos ábrázolásmódból indulok ki. Figyelembe veszem a generált neuronháló terület felhasználását, a kimenet kiszámításának idejét és az eredmény pontosságát.

4.3.1. Lebegőpontos számábrázolás



26. ábra: Mérés, 4 bájtos lebegőpontos adattípust használva.

Utilization Estimates

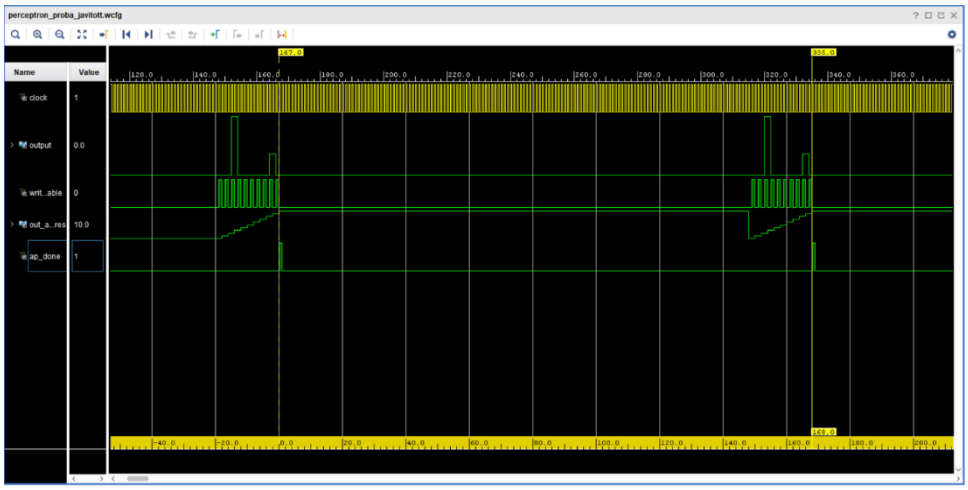
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	442
FIFO	-	-	-	-
Instance	120	50	11710	31830
Memory	-	-	-	-
Multiplexer	-	-	-	59
Register	-	-	1503	-
Total	120	50	13213	32331
Available	120	80	35200	17600
Utilization (%)	100	62	37	183

27. ábra: Erőforrás felhasználási statisztika, 4 bájtos lebegőpontos adattípust használva.

4.3.2. Előjeles fixpontos<32,16>

A számábrázolás során egy számot 32 biten ábrázoltunk, amiből 15 bit az egész rész és a 16 a tört részt és 1 bit az előjel. Ebben az esetben az erőforrásigény a 31. ábra foglalja össze.



28. ábra: Mérés, fixpontos<32,16> adattípust használva.

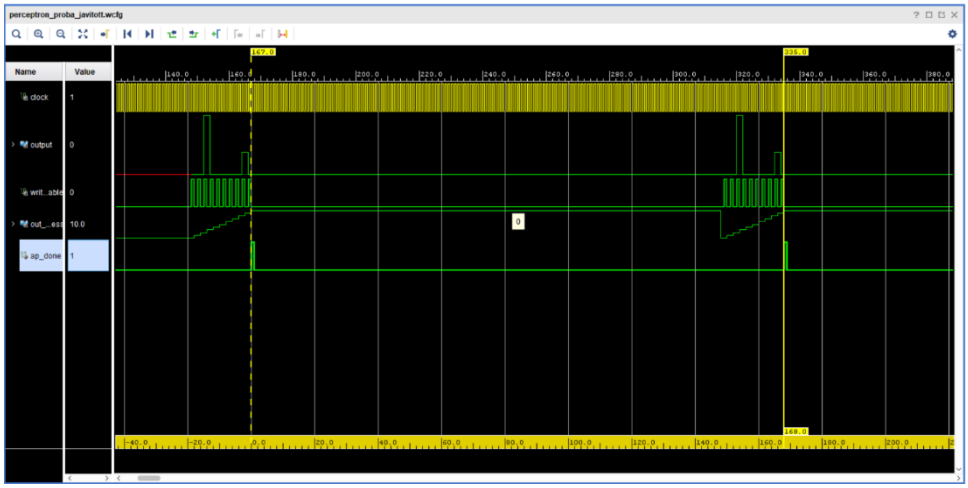
Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	363
FIFO	-	-	-	-
Instance	110	120	10350	34020
Memory	-	-	-	-
Multiplexer	-	-	-	59
Register	-	-	1338	-
Total	110	120	11688	34442
Available	120	80	35200	17600
Utilization (%)	91	150	33	195

29. ábra: Erőforrás felhasználási statisztika, fixpontos<32,16> adattípust használva.

4.3.3. Előjeles fixpontos<16,8>



30. ábra: Mérés, fixpontos<16,8> adattípust használva.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	291
FIFO	-	-	-	-
Instance	70	40	7690	28270
Memory	-	-	-	-
Multiplexer	-	-	-	59
Register	-	-	690	-
Total	70	40	8380	28620
Available	120	80	35200	17600
Utilization (%)	58	50	23	162

31. ábra: Erőforrás felhasználási statisztika, fixpontos<16,8> adattípust használva.

4.3.4. Előjeles fixpontos<12,6>

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	273
FIFO	-	-	-	-
Instance	70	40	6980	27070
Memory	-	-	-	-
Multiplexer	-	-	-	59
Register	-	-	528	-
Total	70	40	7508	27402
Available	120	80	35200	17600
Utilization (%)	58	50	21	155

32. ábra: Erőforrás felhasználási statisztika, fixpontos<12,6> adattípust használva.

4.3.5. Mérések összefoglalása, kiértékelése

A mérések alapján megfigyelhető, hogy fixpontos számábrázolással nagyobb teljesítmény érhető el. A futási idő lebegőpontos számábrázolással 331 órajel, fixpontossal csak 168 órajel. Azonban 4 bájtos fixpontos számábrázolást alkalmazva a rendszer jóval több DSP-t használ fel, mint a 4 bájtos lebegőpontos (float) számokkal. A különböző méretű fixpontos adattípusok nem befolyásoltak a futási időt, viszont kevesebb biten kódolt adatokat használva jelentősen csökken az erőforrásigény. Az adattípus kiválasztásánál a felhasználónak mérlegelni kell, hogy milyen pontosságot szeretne elérni és mennyi erőforrást szeretne felhasználni. Döntési szabadságát korlátozzák a rendelkezésére álló erőforrások száma, valamint a feladat helyes megoldásához szükséges minimális adatméret. Jelen esetben a valós rész minimum 5 biten kell legyen ábrázolva, hogy az eredmény helyes legyen. Azon az értékekre, melyek a (-5,5) intervallumon kívül esnek a szigmoid értéke -1, 1 lesz. 0-5-ig a számok ábrázolására elegendő 4 bit, plusz még szükség van egy előjel bitre. Azok a számok, amelyek nem ábrázolhatóak ennyi biten szaturálódnak. A törtrész 8 biten történő ábrázolásával a kapott legnagyobb hiba a kimeneten 0.0017. Ez a pontosság bőven elégséges a jelenlegi feladathoz. Jelen esetben jó megoldás lenne egy fixpontos<14,5> adattípus használata.

Adattípus	BRAM	DSP48E	FF	LUT	Késleltetés	Legnagyobb hiba
Lebegőpontos	120	50	13213	32331	331	0
fixpontos<32,16>	110	120	11688	34442	168	0.0000075
fixpontos<16,8>	70	40	8380	28620	168	0.0017688
fixpontos<12,6>	70	40	7508	27402	168	0.1783600

33. ábra: Fixpontos és lebegő pontos számábrázolás összehasonlítása [18].

4.4. Radiális bázisfüggvényekből álló hálózat

Radiális bázisfüggvényekből álló hálózatokat (RBF) használnak többváltozós függvények approximációjára, minta felismerésre és osztályozásra, nemlineáris irányítási feladatoknál, rendszerek modellezésére és interpolációra [14].

Az RBF hálók nagy előnye, hogy csak egy rétegből állnak, ezért gyorsak. A háló bemenetei és a bázisfüggvények több dimenziósok is lehetnek. Első lépésben ki van számolva a háló bemenete és

minden bázisfüggvény közepétől való távolsága külön-külön. Ezek az értékek átvezetődnek a bázisfüggényeknek , ami legtöbb esetben a Gauss függvény, de használható akár mexikói kalap függvény is. A bázisfüggvények kimenetei, hasonlóan az előrecsatolt perceptron típusú neurális hálózatokhoz, beszorzódnak a súlytényezőkkel és összesítődnek a háló kimeneti neuronjaiban. A háló tanítható paraméterei a súlytényezők, a bázisfüggvények elhelyezése, valamint az aktivációs függvény paraméterei.

```
    rbf_layer_loop:for(unsigned i=0; i<rbf_number; ++i){
#pragma HLS UNROLL
        MY_FIXED result;
        gauss(IN_local,rbf_center[i],&result);
        rbf_results[i]=result;
    }

    matrix_multiply_rbf(rbf_results,OUT_local,W_rbf,rbf_output,rbf_number);

    for(unsigned i=0;i<rbf_output;++i){
        OUT[step*i]=OUT_local[i];
    }
```

34. ábra: RBF neurális háló.

4.4.1. Távolság (norma számolás)

A bemenet és a bázisfüggvény között levő távolság kiszámolására több lehetőség is van. Amint a [16] kutatásból kiderül a maximum norma (norma maxima) is egy jó megoldás lehet. Én a hagyományosan használt euklidészi norma mellett döntöttem.

$$||x|| = \sqrt{\sum_{i=0}^{n} x_i^2} \qquad \textbf{3.}$$

Euklidészi távolság képlete

Az euklidészi távolság kiszámolásához el kellene végezni a gyökvonást, ami FPGA áramkörön nagy erőforrásigényt eredményezne. Viszont a Gauss képletben ez a norma négyzetre van emelve, kiüti a gyökvonást, ezáltal az euklidészi norma tökéletesen megfelel a célnak.

```
void gauss(const MY_FIXED x[],const MY_FIXED c[rbf_input],MY_FIXED* __restrict__ result){
    MY_FIXED temp, distance=0;
    Calculate_distance_loop:for(unsigned i=0; i<rbf_input; ++i){
#pragma HLS unroll
        temp=x[i]-c[i];
        distance+=temp*temp;
    }
```

35. *ábra: Távolság számolás.*

4.4.2. Gauss bázisfüggvény

A Gauss aktivációs függvény bemenetként megkapja a bázis függvény és az aktuális bemenet közötti távolságot. Kimenet 0 és 1 közötti érték. Minél közelebb helyezkedik el a bemenet a bázisfüggvény középpontjához annál közelebb lesz a Gauss függvény kimenete 1-hez.

$$f(x) = a \cdot \exp\left(-\frac{(x-b)^2}{2c^2}\right) \qquad \textbf{4.}$$

Gauss függvény képlete

Az a paraméter 1-nél nagyobb értéke esetén a kimeneti intervallum növekszik. A b képviseli a bázisfüggvény középpontját, míg a c a Gauss függvény szórását határozza meg.

A Gauss függvényt megvalósítottam szakaszonkénti lineáris approximációval úgy, mint a [15] kutatásban, valamint keresőtáblázatot használva is.

Mivel a távolságok minden esetben pozitív számok ezért mindkét esetben elégséges a Gauss függvény jobb oldalát megvalósítani.

A Gauss függvény paraméterei a keretrendszerben lesznek meghatározva a felhasználó által, majd a keretrendszer generálja a -án levő képlethez szükséges paramétereket.

Szakaszonkénti lineárs megközelítés

```
unsigned distance_uint = distance.range(DATA_WIDTH-1, FRACTION_WIDTH);
if(distance_uint > segments - 1)
{
    *result=0;
}
else
{
    *result = gauss_lut[0][distance_uint] *distance + gauss_lut[1][distance_uint];
}
```

36. *ábra: Szakaszonkénti lineáris megközelítés.*

A szakaszok közötti lépés méretét 1-nek választottam, mert így könnyen megoldható az a és b értékeket tartalmazó tömbök direkt címzése. A fixpontos távolság (distance) egész számmá alakított értékével címzem meg az elsőfokú egyenlet együtthatóit tartalmazó tömböt. A Gauss függvénynek 0-ban van a maximuma. 0-tól távolodva az értéke csökken a szélesség paraméter (szigma) függvényében és egy ponton eléri a 0 értéket. Éppen ezért a keretrendszer csak egy adott számú szakaszt generál 0-tól addig az értékig, amelyiknél a Gauss értéke már megközelítőleg egyenlő 0-val. A segments változó határozza meg a szakaszok számát. Ha a bázisfüggvény és bemenet között számolt távolság egész számmá alakított értéke meghaladja a szegmensek mínusz

egyet, akkor az aktivációs függvény értéke 0-ra állítódik, másképp kiszámoljuk a szakaszonkénti lineáris megközelítéses módszer képlete alapján az értéket.

Keresőtáblázat

```
if(distance>lut_upper_limit)
{
    *result=0;
}
else
{
    int index=distance.range(15,6);
    *result=gauss_Lut[index];
}
```

37. ábra: Keresőtáblázat

Az keresőtáblázatot 0-tól generáltattam, addig az értékig, amire a Gauss kimenete már megközelítőleg 0. Az értékek közötti lépés 2-nek valamely negatív hatványa lesz. Minél kisebb a lépés annál pontosabb lesz a közelítés, de annál több értéket fog tartalmazni a keresőtáblázat, ennek következményeként több erőforrásra lesz szükség. A keresőtáblázatot direkt címzem egy egész értékkel. Ezt az egész értéket fixpontos távolság egész értékeit képviselő bitjeiből, valamint annyi törtrészt képviselő bitből képzem, ahányadik 2 negatív hatvánnyal volt generálva a keresőtáblázat. A képen látható esetben 2^-5-en értékkel volt generáltatva, ezért a 16 bites fixpontos számból kivettem 5 tizedes értéket képviselő bitet 6-tól 11-ig és az egész értékeket képviselő biteket 11-től 15-ig.

4.4.3. Szakaszonkénti lineárs megközelítés és keresőtáblázat módszerek összehasonlítása

A HLS eszközzel szintetizáltam egy 64 bázisfüggvényből álló, 2 dimenziós bemenettel és egy kimenettel rendelkező RBF neuronhálót, különböző direktíva konfigurációkkal ellátva, két aktivációs függvény megvalósítással külön-külön.

Egy 2-es szigmával rendelkező Gauss aktivációs függvényt megvalósítottam hat szegmensből álló szakaszonkénti megközelítéssel és 193 elemből álló keresőtáblázattal is. Azért választottam hat szakaszból álló lineáris megközelítést és 193 elemet tartalmazó keresőtáblázatot, mert a kettővel hasonló pontosság érhető el.

1. táblázat: Lineáris szakaszonkénti megközelítés és keresőtáblázat összehasonlítás

	Átlagos hiba	Legnagyobb hiba
Lineáris megközelítés	0,0128166691	0,046860
Keresőtáblázat	0,0140847591	0,037102

Az átlagos és legnagyobb hibákat a processzoron megvalósított lebegőpontos, exponenciális függvénnyel számolt Gauss értékekhez képest számoltam 289 bemeneti értékre. A modellben 16 bites fixpontos számokat alkalmaztam, ahol 5 bit képviselte az egészrészt és 11 bit a törtrészt.
Az első mérésnél összes ciklust kiterjesztettem teljes mértékben.

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
104	104	104	104	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	59
FIFO	-	-	-	-
Instance	0	189	17712	155420
Memory	-	-	-	-
Multiplexer	-	-	-	59
Register	-	-	1149	-
Total	0	189	18861	155538
Available	120	80	35200	17600
Utilization (%)	0	236	53	883

39. ábra: Keresőtáblázattal

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
107	107	107	107	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	59
FIFO	-	-	-	-
Instance	0	253	22704	84892
Memory	-	-	-	-
Multiplexer	-	-	-	59
Register	-	-	1149	-
Total	0	253	23853	85010
Available	120	80	35200	17600
Utilization (%)	0	316	67	483

38. ábra: Lineáris szakaszonkénti megközelítéssel

A keresőtáblázatos változat csak 3 órajelnyivel lett gyorsabb, 64- el kevesebb DPS-t használ, mert nem kell 1 szorzást és 1 összeadást végezzen minden bázisfüggvénnyel, viszont közel 2-szer annyi LUT-ra van szüksége, mint a lineáris szakaszonkénti megközelítésnek.
A következő mérésnél nem alkalmaztam direktívákat.

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1159	1159	1159	1159	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	359
FIFO	-	-	-	-
Instance	0	1	171	1368
Memory	0	-	13	13
Multiplexer	-	-	-	98
Register	-	-	1210	-
Total	0	2	1394	1838
Available	120	80	35200	17600
Utilization (%)	0	2	3	10

40. ábra: Keresőtáblázattal

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1351	1351	1351	1351	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	359
FIFO	-	-	-	-
Instance	0	2	232	1361
Memory	0	-	13	13
Multiplexer	-	-	-	98
Register	-	-	1210	-
Total	0	3	1455	1831
Available	120	80	35200	17600
Utilization (%)	0	3	4	10

41. ábra: Lineáris szakaszonkénti megközelítéssel

Az keresőtáblázatos változat gyorsabb és meglepő módon kevesebb erőforrást is használt.

A következő mérésnél a neuronok ingerének kiszámítására használt szorzásokat csővezetékesítettem , minden más ciklust teljesen kiterjesztettem.

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
85	85	85	85	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	357
FIFO	-	-	-	-
Instance	0	64	8292	51321
Memory	0	-	13	13
Multiplexer	-	-	-	101
Register	0	-	1321	32
Total	0	65	9626	51824
Available	120	80	35200	17600
Utilization (%)	0	81	27	294

43. ábra: Keresőtáblázattal

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
88	88	88	88	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	357
FIFO	-	-	-	-
Instance	0	128	12196	50873
Memory	0	-	13	13
Multiplexer	-	-	-	101
Register	0	-	1321	32
Total	0	129	13530	51376
Available	120	80	35200	17600
Utilization (%)	0	161	38	291

42. ábra: Lineáris szakaszonkénti megközelítéssel

A keresőtáblázatos változat ebben az esetben is gyorsabb. Ha kiakarjuk terjeszteni a bázisfüggvények kiszámolását végző ciklust a keresőtáblázat egyértelműen a jobb megoldás, mert gyorsabb és a lineáris megközelítés bázisfüggvényenként igényel még egy DSP áramkört pluszba az keresőtáblázatoshoz képes. Az is megfigyelhető, hogy ha csővezetékesítjük a szorzások elvégzését gyorsabb és kevesebb erőforrást igénylő hardvert kapunk, mintha teljesen kiterjesztenénk azokat. Ez azért van, mert a HLS teljes kiterjesztés esetén nem volt képes kialakítani a fa struktúrát szorzások és összeadások elvégzésére, ezért hiába használunk több erőforrást, az adatfüggőség miatt lassúbb lesz a szorzásokat kiterjesztett ciklussal végző változat.

A következő mérésnél a bázisfüggvények és szorzások számolását végző ciklusokat is csővezetékesítettem.

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
145	145	145	145	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	391
FIFO	-	-	-	-
Instance	0	2	282	1895
Memory	0	-	13	13
Multiplexer	-	-	-	140
Register	0	-	1340	64
Total	0	3	1635	2503
Available	120	80	35200	17600
Utilization (%)	0	3	4	14

45. ábra: Keresőtáblázat

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
148	148	148	148	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	391
FIFO	-	-	-	-
Instance	0	3	443	1930
Memory	0	-	13	13
Multiplexer	-	-	-	140
Register	0	-	1344	64
Total	0	4	1800	2538
Available	120	80	35200	17600
Utilization (%)	0	5	5	14

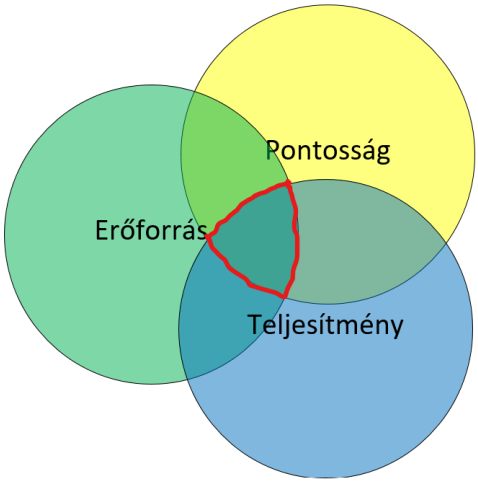
44. ábra: Lineáris szakaszonkénti megközelítéssel

Csővezetékesítést használva is a keresőtáblázatos aktivációs függvényt használó modell gyorsabb, ráadásul kevesebb erőforrást is használt.

Ezen mérések alapján egyértelműen kijelenthető, hogy az én modellem esetében a keresőtáblázatos aktivációs függvény mind erőforrás használat, mind késleltetés szempontjából jobb volt a szakaszonkénti lineáris megközelítéssel megvalósított aktivációs függvénynél.

4.5. Modell optimalizálása

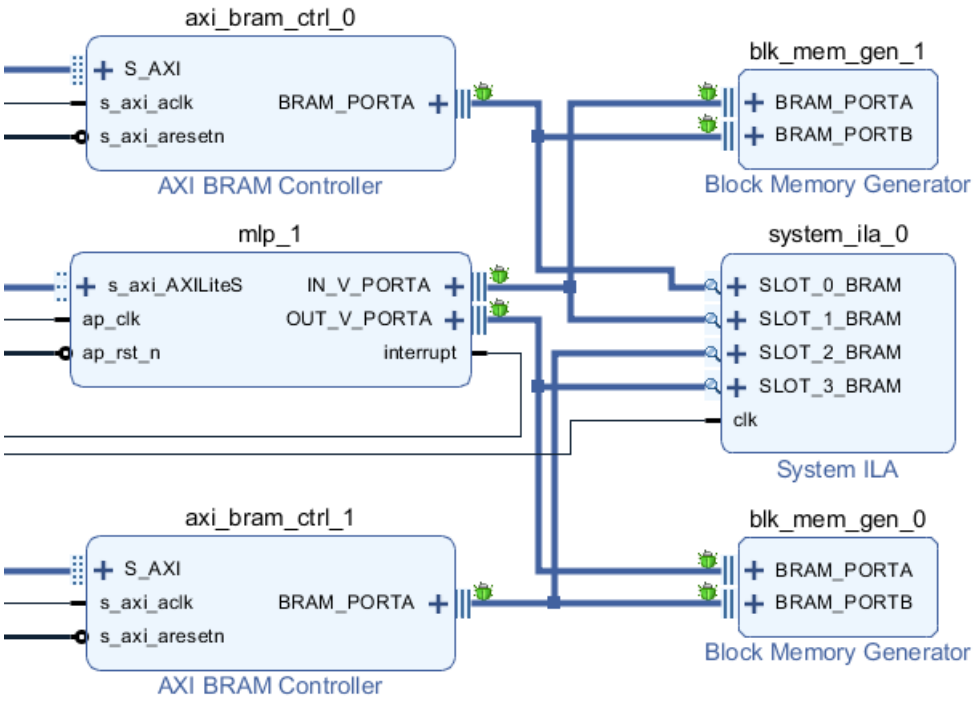
Az optimális modell megalkotásához a tervezőnek mérlegelnie kell a rendelkezésére álló erőforrásokat, az elvárt teljesítményt, valamint a feladat megoldásához szükséges pontosságot. Ha ezek közül bármelyiken javítani szeretne azt egy másik, vagy a másik kettő rovására teheti meg. Ha kettőn szeretne javítani az biztosan a harmadik rovására megy. Az optimális modell eléréséhez a felhasználónak kompromisszumot kell kötnie a három paraméter között. Ahogy az alábbi ábrán is látható, optimális megoldások halmaza a három paraméter metszéspontján belül, a piros vonallal határolt területen található. A keretrendszer célja, hogy a felhasználó számára lehetőséget biztosítson a pontosság és a teljesítmény befolyásolására a rendelkezésre álló erőforrásokhoz mérten.



46. ábra: A modell korlátai.

4.6. A neuronháló interfészelése

Az eddig megvalósított neuronhálóknak (MLP, RBF) egységes interfész van kialakítva. Mindkét neuronhálónak lehet egy vagy több bemenete. A bemenetek értékei a neuronhálón belül tömbökbe vannak eltárolva. A tömbök alapértelmezetten BRAM-al vannak megvalósítva az FPGA-án. A tömböket partícionálom így az értékek regiszterekben lesznek eltárolva, mert a BRAM-ból történő olvasás és írás szűk keresztmetszetet képezne a rendszerben. Az értékek beolvasására ugyanúgy használható BRAM interfész.



47. ábra: Neuronháló interfészelés.

Az 47. ábra-n látható mlp_1 modul a neuronháló. A bemeneteket az SoC-n található ARM processzor az axi_bram_ctrl_0 BRAM kontroller segítségével, amit AXI interfészen keresztül vezérel meg, beolvassa a blk_mem_gen_1 BRAM-ba. Ez a BRAM van összekapcsolva a neuronháló BRAM memóriájával BRAM vezérlő interfészen (AXI BRAM controller) keresztül. A neuronháló a kimeneteit a blk_mem_gen_0 BRAM-ba írja ki, ami az axi_bram_ctrl_1 BRAM kontrollerrel van megvezérelve.

A neuronháló modul interfészeit a HLS eszköz generálja. A generált interfészt direktívákkal lehet befolyásolni.

```
244 #pragma HLS INTERFACE s_axilite port=return
245 #pragma HLS INTERFACE bram port=IN
246 #pragma HLS INTERFACE bram port=OUT
```

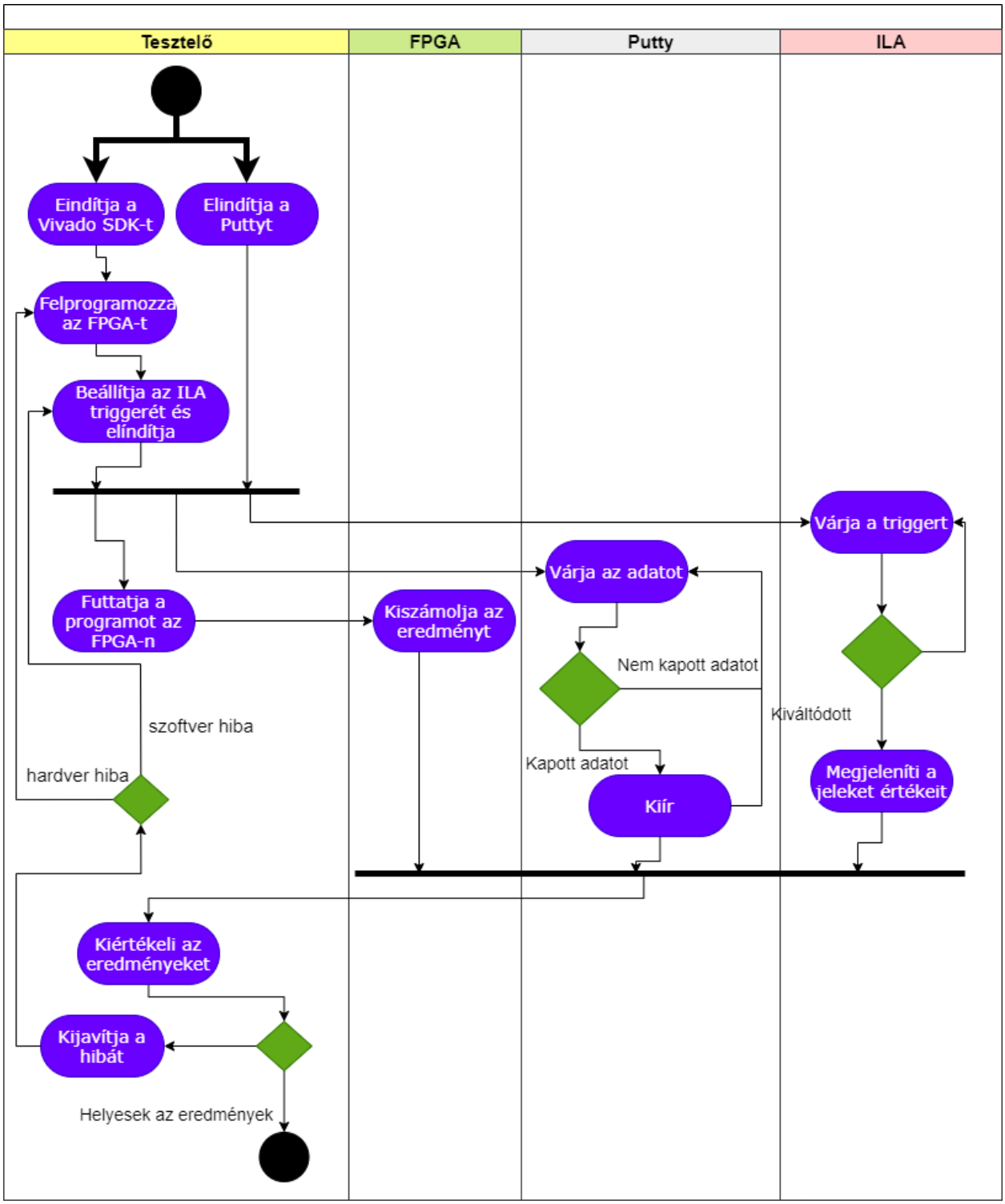
48. ábra: HLS interfész kialakítása direktívákkal.

Az *48. ábra*-n a 244-es sorban látható direktíva a modul vezérlő jeleit axi lite sínrendszerre köti. A következő sorban látható direktíva BRAM interfészt generál a bemenetek beolvasására. A 246-sor BRAM interfészt generál a kimenetek kiírására.

4.7. Hibakeresés (Debuggolás)

A debuggoláshoz az 47. ábra-n látható system_ila_0 nevű integrált logikai analizátort építettük a rendszerbe. A logikai analizátor 0-ás csatlakozójára rávezettük axi_bram_ctrl_0 kimenetét, hogy tesztelni tudjuk, hogy a processzor jó helyre írja be az értékeket és hogy helyesek-e az értékek. Az 1-es csatlakozóra rákapcsoltuk a blk_mem_gen_1 kimeneti portját, ami egyben a neuronháló bemenete azért, hogy ellenőrizzük, hogy a neuronháló jól van-e szinkronizálva a blk_mem_gen_1-al és helyesen olvassa ki az értékeket. A 2-es csatlakozókra a blk_mem_gen_0 A portját kapcsoltuk, ami egyben a neuronháló kimenete. A 3-as csatlakozóra axi_bram_ctrl_1-t kötöttük, hogy ellenőrizzük jó helyről olvassa-e ki a processzor a neuronháló kimeneteit.

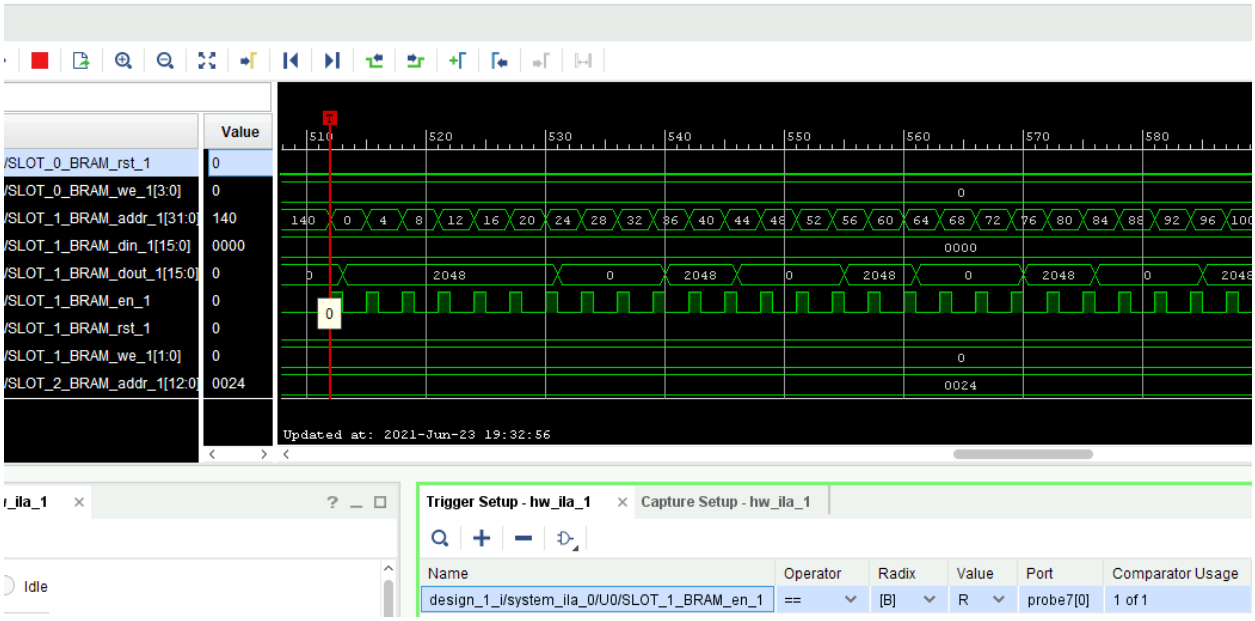
Az 49. ábra-án látható a hibakeresés tevékenység diagrammja. A tesztelést végző személy elindítja a Vivado SDK-t, innen felprogramozza az FPGA áramkört. Beállítja a Vivado ILA ablakában triggert, vagyis kiválasztja, hogy melyik jel milyen értékére, vagy értékének változása pillanatában szeretné elemezni az ILA modulba bevezetett többi jel értékét, majd elindítja az integrált logikai analizátort. Elindítja a putty-t, hogy tudja leolvasni a processzor által soros porton visszaküldött értékeket. Ezután Vivado SDK-ból futtatja a programot az FPGA-n. A processzoron a neuronhálót irányító C program fut, ami feltölti blk_mem_gen_1 memóriát a bemeneti értékekkel, majd neuronháló modul start jelét 1-re állítja. Ezt követően a program aktívan várakozik, amíg a neuronháló az done jelet 1-re állítja, majd kiolvassa a neuronháló által blk_mem_gen_0-ba beírt értékeket és kiíratja azokat Putty terminálon keresztül. Mindeközben a beállított trigger aktiválódik és a Vivado környezetben ellenőrizhetőek a logikai analizátorba bekötött jelek értékei a trigger aktiválódásának pillanatában. A tesztelő kiértékeli az eredményeket. Ha az eredmények nem helyesek, a hiba javítása után újrakezdi a tesztelési folyamatot. Hardveres hiba és módosítás esetén az FPGA újraprogramozásától, szoftveres hiba esetén (C kód) az ILA modul elindításától.



49. ábra: Hibakeresés tevékenység diagram

Az 50. ábra-n látható hogyan ellenőriztük a ILA mag segítségével, hogy a neuronháló helyesen olvassa-e be a bemeneteket blk_mem_gen_1 BRAM-ból. Triggernek a neuronháló által generált

50. ábra: A neuronháló bemenet beolvasásának ellenőrzése ILA-val



enable jel felmenő órajelét állítottuk. Az 50 ábrán a piros kurzor jelzi azt az időpontot, amikor aktiválódott a trigger. Ezután kezdődik a bemenetek beolvasása egyenként. Az első bemenet a bemeneti BRAm memória 0-s címére van írva. A címek 4- el növekednek, mert bár az adatunk csak 2 bájtos a processzor csak 4 bájtónként képes beírni az adatokat a blk_mem_gen_1-ba, ezért az ebből történő olvasást is csak 4 bájtonként lehet helyesen elvégezni. A bemenetek a SoC -ben található ARM processzoron futtatott, az SDK-ban megírt t C kódból vannak megadva. A blk_mem_gen_1 első hat értéke 2048, ami 16 bites fixpontos számábrázolásnál, ahol 11 bit képviseli a törtrészt, 1-nek felel meg. A következő 3 érték 0 és így tovább. Az értékek helyesek és jó sorrendben kerülnek a BRAM kimeneteire és egyben a neuronháló bemenetére.

4.8. A rendszer automatizálása

A Vivado HLS és Vivado szoftverek feladatainak elvégzését egy-egy .tcl szkripttel automatizáltam. A HLS_script.tcl állományt a python keretrendszer Vivado HLS Command Prompt-ban indítja el az `51` - es ábra 3 sorában látható paranccsal.

```
#Vivado HLS Command Prompt
#cd D:/system-main-v4
#vivado_hls -f HLS_script.tcl

cd D:/system-main-v4/mlp-gen
open_project mlp-gen
add_files mlp-gen/Perceptron.h
add_files mlp-gen/Perceptron.cpp
add_files -tb mlp-gen/perceptron_tb.cpp
open_solution "solution1"
set_part {xc7z010clg400-1} -tool vivado
create_clock -period 10 -name default
csynth_design
#cosim_design -setup
export_design -rtl vhdl -format ip_catalog
exec vivado -source D:/system-main-v4/Vivado.tcl
#exit
```

51. ábra: Tcl szkript a Vivado HLS feladatainak automatizálására.

A HLS_script.tcl szkript a Vivado HLS felhaszálói felület elindítása nélkül a háttérben megnyítja a projektet, hozzáadja a python keretrendszer által generált .cpp és .h állományokat a projekthez, megnyitja a kiválasztott solution-t, beállítja a cél eszközt és az órajelet, majd elvégzi a szintetizálást és az ip mag exportálását. Utolsó lépésként elíndítja a Vivado.tcl szkriptet, ami a Vivado feladatait automatizálja.

A szkriptben változtatható paraméterek az elérési utvonalak és céleszköz. Egy HLS projektben több solution hozható létre, hogy a felhasználó egy adott kódhoz többféle direktíva konfigurációt készíthessen a kód megváltoztatása nélkül és összehasonlíthassa az különböző konfigurációk eredményeit. Ennek a módszernek nagy előnye, hogy nem kell módosítani a kódot, hátránya viszont, hogy mindig ki kell választani a kívánt solution-t, amihez tartozik egy directives.tcl nevű állomány. Ez az állomány állítja be a direktívákat, enélkül a direktívák elvesztődnek. Mi egy másik módszert választottunk a különböző direktíva konfigurációk beállítására. A kódba makrókat raktunk azokra a helyekre, ahová direktívát szeretnénk beállítani. A python keretrendszer a makróknak beállítja a felhasználó által kiválasztott direktívákat. Azért, hogy egyszerűsítve legyen a felhasználó dolga, nem a konkrét direktívákat kell beállítsa, csak ki kell választania egyet az előre meghatározott direktíva konfigurációk közül.

```
cd D:/system-main-v4/system-main/valos2
open_project valos2.xpr
update_compile_order -fileset sources_1
open_bd_design {D:/system-main-v4/system-main/valos2/valos2.srcs/sources_1/bd/design_1/design_1.bd}

report_ip_status -name ip_status

set_property  ip_repo_paths  {d:/system-main-v4/system-main/ip_repo/rrtl_modul_1.0 d:/system-main-v4/
mlp-gen/mlp-gen/solution1/impl} [current_project]

update_ip_catalog -rebuild -scan_changes

upgrade_ip [get_ips  design_1_mlp_1_0] -log ip_upgrade.log

export_ip_user_files -of_objects [get_ips design_1_mlp_1_0] -no_script -sync -force -quiet


reset_run synth_1
reset_run design_1_xbar_0_synth_1
launch_runs impl_1 -to_step write_bitstream -jobs 4
```

52. *ábra: Tcl szkript a Vivado feladatainak automatizálására.*

A Vivado.tcl szkript megnyitja az előre elkészített Vivado projektet, megnyítja a projekthez tartozó block design-t ellenőrzi a block design-ba beépített ip magokat, beállítja az ip katalógust a megadott elérési útvonalra, majd frissíti a zip magokat. Ez a frissítés teszi lehetővé, hogy a közvetlenül azelőtt, a HLS által, generált ip mag bekerüljön a projektbe. Végül pedig szintetizálja a projektet, elvégzi az implementálást (fordítás, leképezés, elhelyezés, huzalozás), generálja a bitfájlt és felprogramozza az FPGA áramkört.

4.9. Felmerült problémák és megoldásaik

Az FPGA áramkörre generált teszt modellben 16 bites fixpontos adattípust van alkalmazva. A neuronhálóban a be- és kimenetek tárolására alkalmazott regiszterek 16 bitenként voltak címezve kezdetben. A processzoron futatott SDK programból próbálkoztunk 2 bájtonként beírni a bemeneteket a bemeneti BRAM -ba és 2 bájtonként kiolvasni a kimeneteket a kimeneti BRAM-ból, de a Vivadoban felépített rendszer ezt nem tette lehetővé. A hibát az ILA analizátor segítségével azonosítottuk.

Mivel a neuronhálón kívül eső részben nem volt lehetséges megoldani a problémát, ezért a neuronhálóban változtattuk meg az adatok beolvasásánál és kiírásánál, hogy csak minden második 2 bájtos címről olvasson és írjon.

5. Megoldott feladat

A keretrendszer teszteléséhez megoldottunk egy osztályozási feladatot. Ábrázoltuk a számokat 0-tól 9-ig 7-szer 5-ös mátrixokban többféle képen is, majd felépítettünk 3 neuronhálót, a számok felismerésére. Mindnek 36 bemenete van, 35 a mátrix értékeiből plusz a bias, és 10 kimenete. Mindegyik kimenet a neki megfelelő szám osztályát képviseli. A kimeneti értékek a háló által

meghatározott valószínűségi értéket jelentenek, hogy az illető bemenet mekkora valószínűséggel tartozik az adott kimenetnek megfelelő osztályba. A neuronhálókat ugyanazokkal a tanítási paraméterekkel tanítottuk.

A tesztelt neuronhálók paraméterei

1. 2 rétegű, 36 bemenet és 10 kimeneti neuron
2. 3 rétegű, 36 bemenet, 56 neuronból álló rejtett réteg, 10 kimeneti neuron
3. 5 rétegű, 128, 64, és 32 56 neuronból álló rejtett rétegek, 10 kimeneti neuron

A [19]-ban bemutatott verzióban elvégeztük ugyanezeket a méréseket, viszont akkor a C++ forráskódban még nem volt megvalósítva, hogy a legtöbb neuronból álló rétegén lehessen végig futtatni a neuronháló összes rétegjét és a szigmoid aktivációs függvényt megvalósító keresőtáblázat sem volt direkt címezve. A jelenlegi méréseknél 16 bites fixpontos számokat használtam, a [19]-ban 14 bites fixpontos számok voltak használva. Mindhárom neuronháló esetében a szorzásokat kiszámoló ciklust teljesen kiterjesztettem, valamint a neuronokon iteráló ciklust csővezetékesítettem.

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
263	263	263	263	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	331
FIFO	-	-	-	-
Instance	0	35	8090	28430
Memory	-	-	-	-
Multiplexer	-	-	-	68
Register	-	-	764	-
Total	0	35	8854	28829
Available	120	80	35200	17600
Utilization (%)	0	43	25	163

53. ábra: HLS erőforrás használat és teljesítmény becslés (35->10).

Az első mérés során (53) a HLS 263 órajelciklusra becsülte a rendszer késleltetését. Ha ezt 100 MHz-es órajelen futtatjuk 2,63 µs-ot jelent. A python modell átlagos becsült futási ideje 24.92 µs, ami azt jelenti, hogy a FPGA-n körülbelül 9.5-szörös gyorsítás érhető el.

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
512	512	512	512	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	350
FIFO	-	-	-	-
Instance	0	56	13266	37816
Memory	-	-	-	-
Multiplexer	-	-	-	395
Register	-	-	1521	-
Total	0	56	14787	38561
Available	120	80	35200	17600
Utilization (%)	0	70	42	219

54. ábra: HLS erőforrás használat és teljesítmény becslés (35->56->10).

A második mérés során (54) a HLS 512 órajelciklusra becsülte a rendszer késleltetését, ami 100 MHz-es órajelen futtatva 5,12 µs-ot jelent. A python modell átlagos becsült futási ideje 62.30 µs, tehát körülbelül 12-szeres gyorsítás érhető el. A [19]-ban bemutatott azonos neuronhálón végzett mérésnél 1.68 µs-os késleltetése lett a rendszernek, de annál a mérésnél nem vettük figyelembe, hogy a generált hardver periódusa több, mint 10 ns volt.

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.67	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1278	1278	1278	1278	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	352
FIFO	-	-	-	-
Instance	6890	128	20076	72426
Memory	-	-	-	-
Multiplexer	-	-	-	392
Register	-	-	2654	-
Total	6890	128	22730	73170
Available	3000	3600	1224000	612000
Utilization (%)	229	3	1	11

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.68	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1616	1616	1616	1616	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	352
FIFO	-	-	-	-
Instance	6890	128	25601	74593
Memory	-	-	-	-
Multiplexer	-	-	-	395
Register	-	-	2675	-
Total	6890	128	28276	75340
Available	120	80	35200	17600
Utilization (%)	5741	160	80	428

55. ábra: HLS erőforrás használat és teljesítmény becslés (35->128->64->32->10).

A harmadik mérésnél (55) a generált hardver eddig számunkra ismeretlen okból kifolyólag sok BRAM-ot használ annak ellenére, hogy egyáltalán nem kellen BRAM-ot felhasználjon, ugyanis minden tömb partícionálva van. A baloldali képen ugyanarra a kódra kigenerált teljesítmény és erőforrás becslés látható, csak Virtex 7-es xc7vx980tffg1928-1-es tokozású céleszközt választottam, amely jóval több erőforrással rendelkezik, mint a tesztelésre használt zynq 7-es FPGA áramkör.

6. Eredmények

A mérés során a Python modell Windows 10 operációs rendszeren, i5-7200u processzoron volt mérve, ami megközelítőleg 3.1GHz-n futott. Az FPGA-n való futási időt a HLS teljesítmény becslése alapján számoltam, azzal a feltételezéssel, hogy az FPGA órajele 100MHz.

2. táblázat: A modellek futási ideje

#	**Rétegek száma**	**Futás idő Pythonban (µs)**	**Futás idő FPGA-n (becsült) (µs)**	**FPGA-n elért gyorsítás**
1	2 (35, 10)	24.92	2,63	9,47
2	3 (35, 36, 10)	62.30	5,12	12.16
3	5(35, 128, 64, 32, 10)	112.21	16.16	6.94

3. táblázat: A modellek tanítási és futási ideje a [19] dolgozatból.

#	**Rétegek száma**	**Tanítható paraméterek**	**Tanítási idő (s)**	**Futás idő Pythonban (µs)**	**Futás idő FPGA-n (becsült) (µs)**
1	2 (35, 10)	360	50	72	1,68
2	3 (35, 36, 10)	1666	67	319	29,51
3	5(35, 128, 64, 32, 10)	31786	107	606	207,09

4. táblázat: A modellek négyzetes hibáinak az átlaga a [19] dolgozatból.

#	Rétegek száma	Python modell négyzetes hibájának az átlaga	HLS modell négyzetes hibájának az átlaga	Modellek közötti eltérés
1	2	0.0401977984465673	0.0398001911983511	0.0052413841091167
2	3	0.0213411106963915	0.0205169712748613	0.0054848397495561
3	5	0.0210445014637294	0.0231401516365874	0.0107506224136835

Amint látható a 4. táblázatban, annak ellenére, hogy a szimulált HLS modellben az adat típus sokkal kisseb és egyszerűbb a két megvalósítás közti négyzetes eltérésének az átlaga csak 1%-os. Meglepő módon az első két mérésnél a HLS modell átlagos négyzetes hibája kisebb lett, mint a Python modellé.

6.1. Hasonló rendszerekkel való összehasonlítás

6.1.1. Saját rendszer összehasonlítva LeFlow-al

Az összehasonlítást egyelőre csak elméleti szinten tudtam elvégezni, mivel a Microchip felvásárolta a LeFlow rendszer által használt LegUp magas szintű szintetizáló eszközt, amit a Torontói kutató egyetemen fejlesztettek ki [17].

Előnyök

A saját rendszerünk támogat fixpontos, integer és lebegőpontos számábrázolást is, míg a LeFlow csak lebegőpontost.

Dírekt FPGA-ra tervezett párhuzamosított mikróarchitektúrák generálódnak, amelyek elméletileg hatékonyabbak kellene legyenek, mint a LeFlow által használt XLA kompilátor által CPU-kra generált architektúrák.

Hátrányok

Korlátozott számú támogatott neuronháló típus és aktivációs függvény elérhető a saját rendszerünkhöz. Ezzel szemben a LeFlow a TensorFlow által generált modellek nagy részét képes lefordítani FPGA áramkörre.

6.1.2. A CPU-n futatott python modell összehasonlítása az FPGA-n futatott modellel

A HLS-ben és Python-ban, azonos neuronhálóval és azonos bemenetekkel végzett mérések alapján megfigyelhető, hogy a [19]-ban elért mérési eredményekhez képest mind a python mind a HLS modellt jelentős mértékben tudtuk javítani. FPGA áramkörön elért gyorsítás jelentős közel 9.5-szörös a legkisebb neuronhálóval végzett mérés esetében. A második nagyobb méretű neuronhálónál még nagyobb gyorsítás látható több mint 12-szeres, míg a legnagyobb neuronhálónál a gyorsítás csak 6.94-szeres a processzoron futtatott Python-ban felépített modellhez képest. A harmadik neuronhálónál sok BRAM-ot használ a generált hardver, annak ellenére, hogy egyet sem szabadna felhasználjon, mivel az összes tömböt partícionáltam. Ennek a jelenségnek a kivizsgálása és kiküszöbölése további kutatást és fejlesztést igényel.

7. Következtetések

7.1. Megvalósítások

A kutatást az Accenture Student Research Scholarship pályázat keretén belül kezdtük el és azóta folyamatosan fejlesztjük a rendszert. A projektet sikeresnek tekintem, mert:

- Tanulmányoztam a LeFlow keretrendszert elméleti szinten, felismerve annak néhány hiányosságát és gyengeségét. Ezeket a gyengeségeket saját rendszerünkben igyekeztem pótolni.
- Megvalósítottam paraméterezhető MLP és RBF neuronhálókat különböző aktivációs függvényekkel a HLS magasszintű szintetizáló eszközt alkalmazva.
- A megvalósított neuronhálókat HLS-ben tesztpadok által ellenőriztem, az eredményeket összehasonlítottam a csapattársam által készített Python modell eredményeivel.
- Különböző háló mikróarchitektúrákat alakítottam ki direktívák segítségével, a megoldásokat összehasonlítottam egymással a késleltetés és erőforráshasználat szempontjából.
- A rendszert teszteltem különböző számábrázolásokat alkalmazva és az eredményeket kiértékeltem pontosság, késleltetés és erőforrás használat szempontjából.
- A Gauss bázisfüggvényt megvalósítottam keresőtáblázat és szakaszonkénti lineáris megközelítéssel. A két módszert mérések során összehasonlítottam és kiértékeltem.
- Kezdetben, megvalósított interfész hiányában, a rendszert Simulink-ben építettem ki és System Generator használatával teszteltem.

- A végső rendszert Vivadoban építettük ki. A bemeneteket ARM processzoron keresztül beírjuk egy BRAM-ba, ahonnan a neuronháló kiolvassa őket, kiszámolja a kimeneteket, kiírja egy másik BRAM-ba, innen pedig a processzor kiolvassa az eredményeket.
- A Vivadoban felépített rendszerbe integrált logikai analizátort helyeztünk, hogy a különböző interfészek csatlakoztatásánál fellépő hibákat megértsük és kitudjuk javítani.
- Automatizáltam a HLS szintézis folyamatát, a HLS ip mag integrálását a Vivadoban kiépített rendszerbe és a bit fájl generálását Vivado-ban.
- Felépítettünk egy automatikusan működő rendszert, amit valós hardveren is kipróbáltunk.
- Az általunk épített rendszerrel megoldottunk egy osztályozási feladatot.
- Megoldott problémával valós hardveren teszteltük a rendszert, az eredményeket összehasonlítottuk a processzoron futatott modellel.
- A rendszerünket elméleti szinten összehasonlítottam a LeFlow rendszerrel.

A projekt során felmerülő problémák, nehézségek:

- Egyik nehézség a sok különböző eszköz használatának megtanulása, és a különböző részek egy rendszerbe történő beillesztése volt.
- Felhasznált eszközök és programozási nyelvek: Python, C++, C, Vivado HLS, Vivado, Matlab, Simulink, System Generator, Vivado SDK, Tcl szriptek.
- A HLS eszköz szintézis folyamata, amely során a hardver leírást generálja a C++ kódból, csak egy bizonyos szintig befolyásolható, nem mindig képes a megadott direktívák alapján generálni a hardvert. Sokszor kiszámíthatatlannak tűnő hardverleírásokat generál, és nehéz rájönni, hogy miért.
- Nehézség volt a hardveren történő fejlesztéseket online kivitelezni, ugyanis nem tartózkodhattunk a céleszköz mellett, a labortechnikus szükség esetén újraindította az FPGA lapot.

7.2. További célkitűzések

- A neuronhálók FPGA áramkörön történő tanítása.
- Súlytényezők működés közben való beolvasása. A súlytényezők legyenek változtathatóak működés közben a neuronháló újragenerálása nélkül.
- Szeretnék további neuronháló modelleket kiépíteni: Kohonen, CMAC, időfüggő neuronhálók. Végső célunk a napjainkban népszerű konvolúciós neuronháló megvalósítása.

- LeFlow rendszer kipróbálása és az általa generált modellek összehasonlítása a saját rendszerünk által generált modellekkel késleltetés és erőforrás használat szempontjából.
- Saját rendszer összehasonlítása más neuronhálókat hardveresen gyorsító rendszerekkel (Intel® Movidius™ Vision Processing Units, GPU stb.).
- A HLS modellek gyorsítása, optimalizálása.
- Más interfészek kiépítése a rendszerhez: hálózati interfész, HDMI csatlakozó.
- A rendszert felhasználni egy mobilis robot FPGA áramkörön megvalósított képfelismerő funkciójának megvalósítására.
- MicroBlaze processzorral is megvalósítani a rendszert, hogy olyan áramkörökön is futtatható legyen, amelyek nem rendelkeznek ARM processzorral.

7.3. Köszönetnyilvánítás

- Köszönöm szépen a bíztatást, útbaigazítást és a rengeteg jó tanácsot vezetőtanáromnak, dr. Brassai Sándor Tihamérnek.
- Köszönöm szépen Székely István Zsolt labortechnikusnak, hogy mikor megkértük, alkalmazkodott hozzánk és segített a hardver csatlakoztatásával a laborgépekhez, valamint felügyelte az általunk online használt eszközöket.
- Köszönöm az Accenture-nek, hogy az „Accenture Student Research Scholarship 2020” pályázat keretén belül támogatták a dolgozatomat.

8. Irodalomjegyzék

[1] Van Gerven, M. and Bohte, S., 2017. Artificial neural networks as models of neural information processing. Frontiers in Computational Neuroscience, 11, p.114.

[2] Liang, S., Yin, S., Liu, L., Luk, W. and Wei, S., 2018. FP-BNN: Binarized neural network on FPGA. Neurocomputing, 275, pp.1072-1086.

[3] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang and J. Cong, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE, 2017, pp. 152-159

[4] Coussy, P., Gajski, D.D., Meredith, M. and Takach, A., 2009. An introduction to high-level synthesis. IEEE Design & Test of Computers, 26(4), pp.8-17.

[5] O'Loughlin, D., Coffey, A., Callaly, F., Lyons, D. and Morgan, F., 2014. Xilinx vivado high level synthesis: Case studies.

[6] Xilinx, 2021, HLS Key Documents and Getting Started 23 április 2021, <https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/HLS-Key-Documents-and-Getting-Started-FAQ/td-p/1118001>

[7] Xilinx, 2018, HLS Pragmas, 23 április 2021, <https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/okr1504034364623-1.html>

[8] Lahti, S., Sjövall, P., Vanne, J. and Hämäläinen, T.D., 2018. Are we there yet? A study on the state of high-level synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 38(5), pp.898-911.

[9] Goldsborough, P., 2016. A tour of tensorflow. arXiv preprint arXiv:1610.01178.

[10] Noronha, D.H., Gibson, K., Salehpour, B. and Wilton, S.J., 2018, December. Leflow: Automatic compilation of tensorflow machine learning applications to fpgas. In 2018 International Conference on Field-Programmable Technology (FPT) (pp. 393-396). IEEE.

[11] Noronha, D.H., Salehpour, B. and Wilton, S.J., 2018, August. LeFlow: Enabling flexible FPGA high-level synthesis of tensorflow deep neural networks. In FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers (pp. 1-8). VDE.

[12] Kastner, R., Matai, J. and Neuendorffer, S., 2018. Parallel programming for FPGAs. arXiv preprint arXiv:1805.03648.

[13] Finnerty, A. and Ratigner, H., 2017. Reduce power and cost by converting from floating point to fixed point. WP491 (v1. 0).

[14] Brassai, S.T., 2019. Neurális hálózatok és fuzzy logika, Scientia kiadó.

[15] Shymkovych, V., Telenyk, S. and Kravets, P., 2021. Hardware implementation of radial-basis neural networks with Gaussian activation functions on FPGA. Neural Computing and Applications, pp.1-13.

[16] Brassai, S.T., Bako, L., Pana, G. and Dan, S., 2008, May. Neural control based on RBF network implemented on FPGA. In 2008 11th International Conference on Optimization of Electrical and Electronic Equipment (pp. 41-46). IEEE.

[17] https://www.microchip.com/en-us/about/news-releases/products/microchip-acquires-high-level-synthesis-tool-provider-legup

[18] B. Bustya, A. Hammas, „Keretrendszer MLP-háló FPGA-alapú megvalósítására,” in XX. Online Kari Tudományos Diákköri Konferencia, Marosvásárhely, 2021.

[19] B. Bustya, A. Hammas, „Keretrendszer neurális háló FPGA alapú megvalósítására,” in XXII. Műszaki Tudományos Diákköri Konferencia, Temesvár, 2021.

UNIVERSITATEA SAPIENTIA DIN CLUJ-NAPOCA

FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE, TÎRGU-MUREȘ
SPECIALIZAREA CALCULATOARE

Vizat decan

Ș.l. dr. ing Kelemen András

Vizat director departament

Conf. dr. ing. Domokos József