Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Rok Bricman

Realizacija mehkega inferenčnega stroja v okolju CUDA

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: doc. dr. Iztok Lebar Bajec

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V diplomski nalogi razvijte programsko opremo za realizacijo mehkega inferenčnega stroja v okolju CUDA. Pri tem se osredotočite na mehki sistem z dvema vhodnima spremenljivkama in eno izhodno. Implementacijo testirajte in primerjajte z referenčno serijsko implementacijo. Rezultate kritično komentirajte.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Rok Bricman, z vpisno številko **63040015**, sem avtor diplomskega dela z naslovom:

Realizacija mehkega inferenčnega stroja v okolju CUDA

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Iztoka Lebarja Bajca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 01. septembra 2014 Podpis avtorja:

Zahvala

Zahvaljujem se mentorju, prijateljem in sorodnikom za podporo in potrpežljivost.

Kazalo

Seznam uporabljenih kratic in izrazov

GPE - grafična procesna enota (angl. GPU - graphics processing unit)

CPE - centralna procesna enota (angl. CPU - central processing unit)

GPGPU - splošno namensko računanje z uporabo grafične strojne opreme (General Purpose Computation Using Graphics Hardware)

API - aplikacijski programski vmesnik (Application Programming Interface)

zvitek niti - skupina (trenutno dvaintridesetih) implicitno sihroniziranih niti na GPE (angl. warp)

jedro - funkcija, ki jo vzporedno izvajajo niti na GPE (angl. kernel)

FIS - sistem mehke inference (angl. fuzzy inference system)

FLD - podatkovna množica orodja fuzzylite (angl. fuzzylite dataset)

Povzetek

Vse procese programsko realiziranega inferenčnega stroja imenujemo mehko procesiranje. Ti procesi so mehčanje vhodnih vrednosti, mehka inferenca ali sklepanje in ostrenje mehke množice, v tem vrstnem redu. Sama izvedba se imenuje mehki inferenčni stroj ali sistem mehke logike. V diplomskem delu je opisan takšnen sistem v vzporedni izvedbi po posebnem pristopu, kjer v primerjavi z zaporedno izvedbo kompleksnejše operacije razbijemo na več preprostejših. Za izvedbo smo uporabili arhitekturo CUDA, ki nam omogoča splošno namensko vzporedno računanje s sodobnimi GPE-ji. To izvedbo smo na koncu testirali ter primerjali z zaporedno izvedbo na CPE-ju. Primerjali pa smo tudi natančnost rezultatov in čase operacij algoritmov. Čas je merjen posebej za vse tri glavne procese. Prav tako je merjen tudi skupni čas, ki zajema tudi deklaracije, rezerviranje in kopiranje v pomnilnik. Podatki vhodnih vrednosti in baz znanj so pripravljeni in pridobljeni iz programskih paketov MATLAB in fuzzylite.

Ključne besede: mehko procesiranje, mehka inferenca, mehčanje, ostrenje, GPE procesiranje, CUDA.

Abstract

All processes in a software implemented inference machine are called fuzzy processes. These processes are fuzzification, fuzzy inference and defuzzification, executed in that order. The implementation itself is called a fuzzy inference machine or a fuzzy logic system. This thesis describes such a system in a parallel implementation with a specific approach, where complex operations are broken down into multiple simpler ones. We used the CUDA architecture, which allows us the usage of general purpose parallel computing on modern GPU's. At the end we tested this implementation, in comparison with the sequential implementation on the CPU, by comparing the precision of the computational results and the needed times of operation algorithms.

Key words: fuzzy processing, fuzzy inference, fuzzification, defuzzification, GPU processing, CUDA.

Poglavje 1

Uvod

Živimo v času, ko imamo iz dneva v dan opravka z vse večjimi zbirkami podatkov in objektov digitalnih struktur. Na drugi strani pa imamo razvijalce strojne opreme, predvsem centralno procesnih enot (v nadaljevanju CPE), ki s temi zahtevami več ne shajajo in tako prihaja do vedno daljših časov pri obdelovanju teh podatkov. Problem? Ga ni, ker (na srečo) že imamo tehnologijo in metode za alternativno reševanje teh problemov.

Ena izmed trenutno najuspešnejših metod je izkoriščanje grafičnih procesnih enot (v nadaljevanju GPE), katerih razvoj je v zadnjih letih zelo napredoval. Njihova prednost pred enotami CPE je veliko število jeder za splošno namensko računanje, ki s paralelnostjo dosežejo neverjetno hitro procesirno hitrost SIMD arhitekture, imenovano SIMT (angl. Single Instruction Multiple Thread). Da lahko GPE uporabimo na tak način, potrebujemo posebno platformo združbe NVIDIA, imenovano CUDA (angl. Compute Unified Device Architecture), ki nam omogoča paralelno podatkovno preračunavanje brez zahteve in potrebe po znanju grafičnih API-jev.

V tej diplomski nalogi bomo z omenjeno metodo poskušali izboljšati čase procesiranja ene izmed metod preračunavanja, ki se prav tako vse več uporablja na večjih množicah števil in jo imenujemo mehka inferenca. Mehka inferenca je del mehke logike, v kateri imamo v nasprotju s trdo logiko namesto dveh neskončno število stanj in namesto s posameznimi trdimi vre-

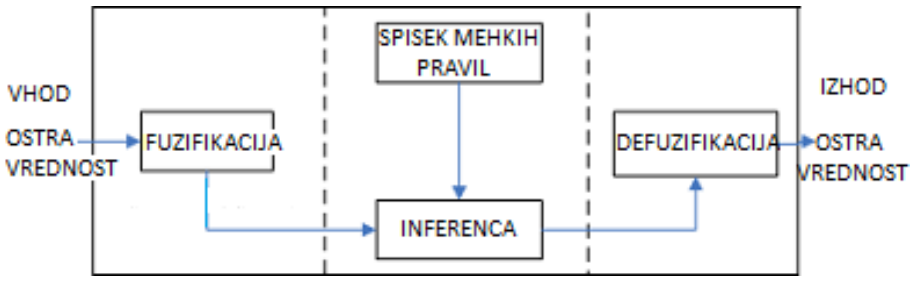
dnostmi se ukvarjamo z mehkimi množicami.

Tako bomo implementirali preprost Mamdanijev sistem mehke logike tipa 1 v arhitekturi CUDA, ki ji bomo podali spisek mehkih pravil, vrednosti dveh vhodnih spremenjivk, in bomo iz njega dobili vrednost enega izhoda. Tega bomo nato testirali v primerjavi z izvedbo v CPE-ju, v natančnosti preračunavanja in v primerjavi s samim časom izvajanja.

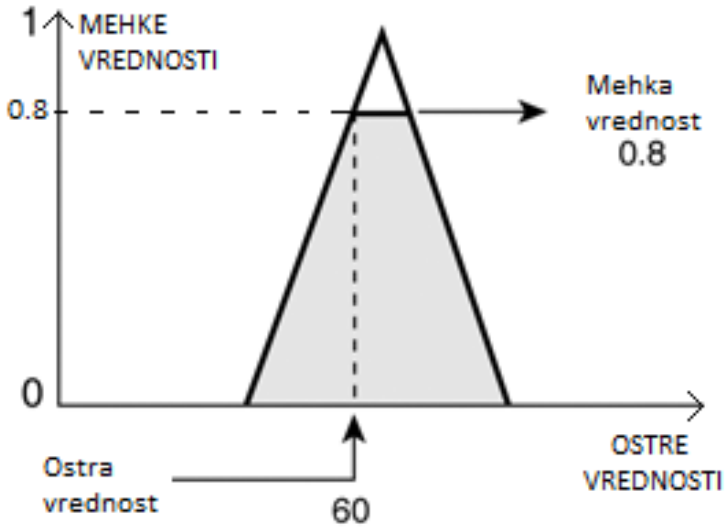
Poglavje 2

Mehki sistem

Mehki sistem je sistem, ki uporablja temeljna načela mehke logike. Zanj so značilne nejasne, dvoumne, nenatančne, popačene ali celo manjkajoče informacije. Na podlagi teh informacij ter določenega spiska pravil poda končen sklep[1]. Proceduro sistema v grobem delimo na tri procese. Glavni proces, ki opravlja zgoraj opisano nalogo, imenujemo inferenčno procesiranje. Pred inferenčnim procesiranjem imamo bolj ali manj izrazit proces mehčanja. Po inferenčnem procesiranju pa nastopi proces ostrenja. Procesor mora obdelati vse tri v zaporedju, kot smo ga napisali in je tudi prikazano na sliki 2.1. Sistem lahko realiziramo po več modelih. Eden od manj zapletenih in široko uporabljenih je Mamdanijev model, za katerega je značilno, da sta pogoj in posledica vsakega pravila mehki vrednosti. Nekateri modeli, npr. Takagi-Sugeno, delujejo po podobnem principu, le da imajo pogoje in posledice sestavljene iz mehkih in ostrih vrednosti.



Slika 2.1: Shema korakov sistema mehke logike.



Slika 2.2: Primer fuzifikacije vhodne vrednosti.

2.1 Proces mehčanja

Mehčanje je postopek za prirejanje in doseganje mehkosti vhodnih vrednosti po konceptu teorije mehkih množic[3]. Vhodna vrednost je torej ostra, prirediti pa ji je potrebno primerno mehkost, da bo lahko pravilno upoštevana v procesu inference. To naredimo tako, da napravimo presek med mehko množico in ostro vhodno vrednostjo, kot prikazuje slika 2.2. Tako se vrednostim vhodnih spremenljivk določi, v katero mehko množico spadajo po vplivnih faktorjih spiska pravil.

$$\frac{\sum_{i=1}^{N} x_i m(x_i)}{\sum_{i=1}^{N} m(x_i)},$$

Slika 2.3: Enačba težiščne metode.

2.2 Mehka inferenca

Mehka inferenca običajno temelji na agregaciji, implikaciji in akumulaciji. Agregacija je potrebna, da več pogojev sestavimo v en združen pogoj[1]. Implikacija ali neposredno sklepanje nam omogoča prehod iz IF-dela v THEN-del mehkega pravila, za katerega se pogosto uporablja Mamdanijev model, ker za implikacijo zahteva le operacijo `min()`. Takšno inferenčno shemo poznamo pod imenom MAX-MIN. Na koncu imamo še akumulacijo, ki skrbi za pravilno sestavo rezultativnih komponent v eno skupno mehko množico in jo ponavadi implementiramo z operacijo `max()`.

2.3 Proces ostrenja

Brez ostrenja bi končni rezultat po fazi sklepanja ostal mehka množica. V tem koraku se mehka množica zmanjša na eno ostro vrednost izhoda. Obstaja več vrst metod ostrenja, ki se izbirajo glede na primernost danega problema. Tukaj pa bi predvsem omenili težiščno metodo (angl. Center Of Gravity), ki sodi med najbolj popularne metode z veliko frekvenco uporabe. Največja slabost te metode v sistemih je, da potrebuje numerično procesiranje števca in imenovalca, ki ne zajema preprostih operacij kot sta `min()` in `max()`, kar pa ne velja, če za tak sistem uporabljamo CPE. Kot vidimo v enačbi 2.3, se ta metoda v nadaljevanju še poenostavi, če je funkcija diskretna.

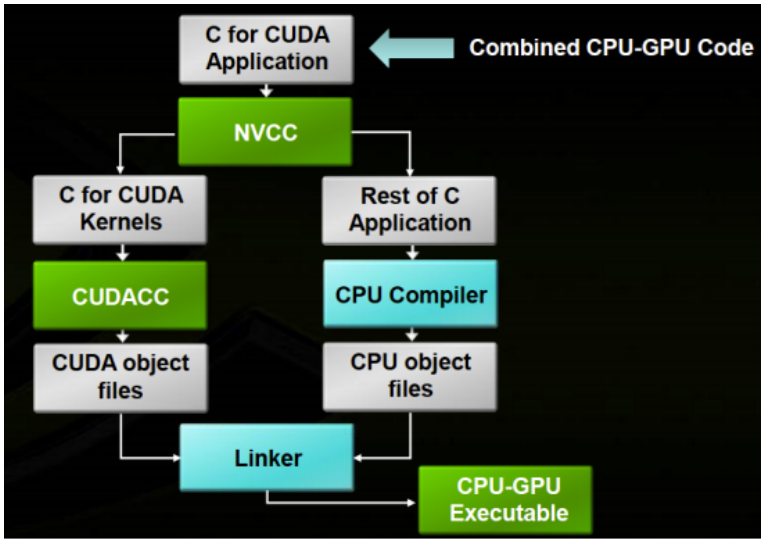
Poglavje 3

CUDA

CUDA (angl. Compute Unified Device Arhitecture) je arhitektura za splošno namensko vzporedno računanje. Takšen pristop v širšem imenujemo tudi GPGPU (angl. General-purpose computing on graphics processing units), ki temelji na velikem številu niti, ki se izvajajo počasneje, namesto da bi izvajali samo eno nit zelo hitro.

Arhitekturo uporabljamo na izbranih GPE-jih združbe NVIDIA, katera jo je tudi razvila. Z njenim API-jem dobimo stopnjo abstrakcije strojne opreme brez neposrednega vpogleda na celotno arhitekturo v ozadju[7].

Njene komponente so razporejene v dveh skupinah[7]. CPE in sistemski pomnilnik so del zunanjih komponent in njihovo skupino imenujemo gostitelj (angl. host). Vse ostale komponente, ki tvorijo GPE arhitekturo, se nahajajo na napravi in spadajo v skupino naprava (angl. device). Med njima imamo vmesnik, ki skrbi za komunikacijo (kot so odzivi na ukaze) in olajša prenos podatkov. Podrobnosti posameznih komponent naprave ne bomo opisovali natančneje, ker so njihovi opisi in hierhija podani že v drugih virih[7]. V nadaljevanju omenimo le SM (angl. streaming multiprocesor), ki je multiprocesor z osmimi SP-ji (angl. streaming processors), ki izvajajo vse procese na napravi.

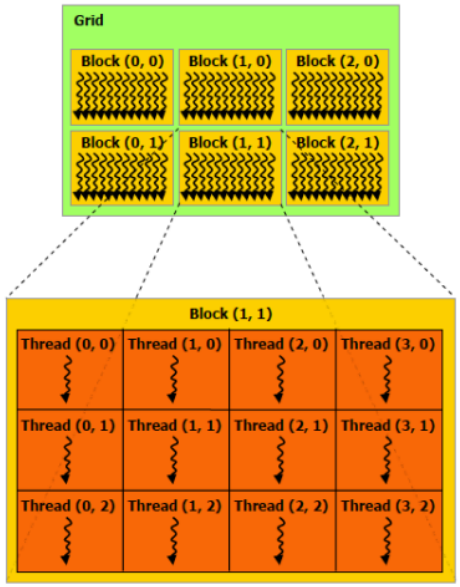


Slika 3.1: Proces prevajanja izvorne kode CUDE.

3.1 Programski model v CUDI

Pri programiranju v CUDI v glavnem delamo z vzporednem preračunavanju na napravi. Še vedno pa potrebujemo asistenco z strani gostitelja pri upravljanju in nadzorovanju celotnega programa. Tako poganjamo izvorno kodo na dveh različnih platformah: gostitelj in naprava[10]. Na začetku je izvorna koda sestavljena iz kode gostitelja in naprave v eni datoteki. NVIDIIN C-prevajalnik (nvcc) najprej razčleni izvorno kodo na dva dela, ki se za vsako platformo izvršita ločeno. Za razčlembo potrebujemo ključne besede pred ukazi, ki so namenjeni izvajanju na napravi. Koda gostitelja se prevede s standardnim C/C++ prevajalnikom, ki ustvari standardne datoteke objektov. Koda naprave se prevede s CUDA C prevajalnikom (CUDACC), ki ustvari datoteke objektov CUDA. Oba sklopa objektov sta na koncu povezana v en skupno izvršljiv program, kot je prikazano na sliki 3.1[9].

Jedra predstavljajo samo srce programa v CUDI in jih v kodi vidimo, kot normalne C funkcije, z razliko da so definirane s ključno besedo `__global__`. Ker vse niti v jedru istočasno izvajajo isto kodo, imenujemo to paradigmo SPMD (Single Program Multiple Data), kar pomeni en program in več podat-



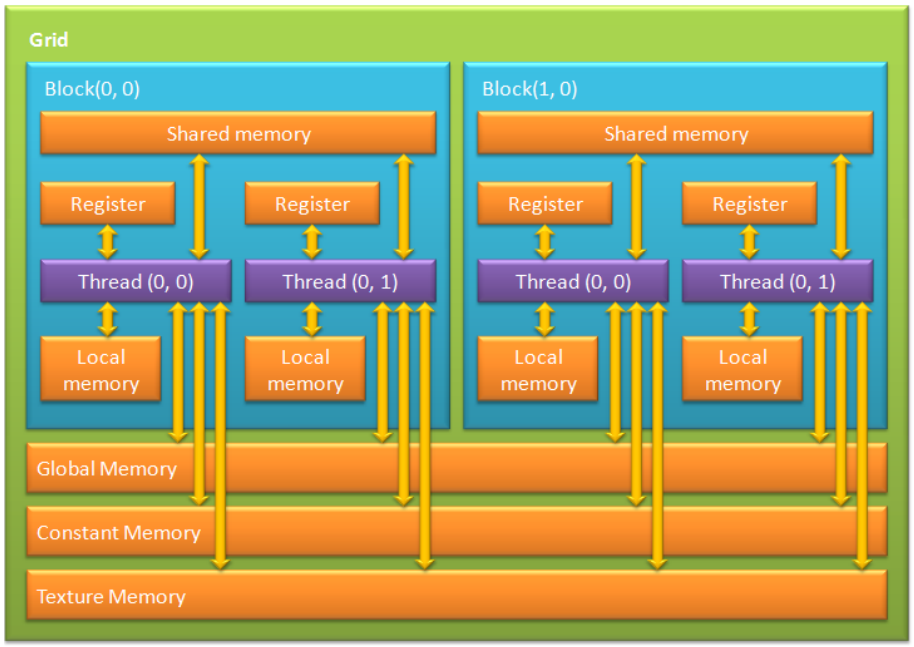
Slika 3.2: Mreža blokov niti.

kov, ki je široko uporabljen pojem v vzporednem preračunavanju[7]. Vsaka nit, ki se izvaja, dobi dodeljeno edinstveno ID (identifikacijsko število) niti. Tako se lahko posamezna nit v jedru identificira s kombinacijo svojih blockIdx, blockIdx in threadIdx vrednosti. Na napravi se lahko naenkrat izvaja samo eno jedro.

3.2 Hierarhija niti

Niti na napravi se kličejo samodejno, ko se izvede jedro. Po konceptu so zelo podobne nitim CPE-ja, kjer ima vsaka nit svoj edinstveni ID in svoj lokalni pomnilnik (registre). Programer določi število niti skupaj z njihovo konfiguracijo, ki se bodo izvajale v jedru. Vse niti v izvedbi jedra, imenujemo mreža (angl. grid) 3.2[10].

Mrežo sestavlja eden ali več blokov niti. Blok je niz niti, ki istočasno izvajajo isti algoritem. Niti v bloku lahko med sabo sodelujejo pri doseganju



Slika 3.3: Različni tipi pomnilnikov naprave.

skupnih rezultatov. S tem pripomoremo, da niti razdelimo v manjše skupine, kar olajša sihronizacijo med njimi. Blok niti lahko razdelimo eno-, dve- ali tri-dimenzionalno. Vsak blok ima svoj edinstven identifikator bloka. Vse niti v njem lahko med seboj sodelujejo in izmenjujejo podatke shranjene v skupnem (angl. shared) pomnilniku. Za sihronizacijo niti v bloku uporabimo ukaz `__syncthreads()`.

3.3 Pomnilnik v CUDI

Za uporabo pomnilnika na napravi, moramo nanj prvo prekopirati podatke iz sistemskega pomnilnika. Tako ob izvršitvi ukaza v jedru, niti pridobijo dostop do podatkov v njem. Dostop je lahko počasen in ima omejeno pasovno širino, kar pri več tisoč nitih povzroča ozko grlo. Za olajšanje tega problema imamo v CUDI na voljo več vrst pomnilnikov, z pravilno izbiro katerih, izboljšujemo učinkovitost izvajanja.

Obstaja več vrst pomnilnika naprave, nekatere so: globalni, konstantni, teksturni, skupni in registri (kot je prikazano na sliki 3.3[8]). V glavnem se razlikujejo po velikosti, času dostopa in stopnji privatnosti (omejitvi dostopa).

- Globalni pomnilnik je največji po velikosti in ima najvišji čas dostopa med vsemi. V API-ju CUDE je najlažji za uporabo in zahteva le poznavanje nekaj ukazov. Za rezervacijo prostora uporabimo funkcijo `cudaMalloc`. Za kopiranje podatkov iz sistemskega pomnilnika v globalnega še potrebujemo funkcijo `cudaMemcpy`. Po preračunavanju uporabimo enak korak pri kopiranju podatkov nazaj na sistemski pomnilnik. Na koncu dodeljen prostor pomnilnika sprostimo s funkcijo `cudaFree()`. Podatki v pomnilniku se ohranijo za čas trajanja celotnega programa in so dostopni za vsako nit v katerikoli mreži. Tako ga uporabljamo za shranjevanje podatkov iz sklicev enega jedra, ki jih uporabimo pri sklicevanju naslednjega in pri izmenjavi podatkov med mrežami in bloki.

- Konstantni pomnilnik je zasnovan tako, da ga med izvajanjem jedra lahko samo beremo. To omogoča hitrejši dostop pri vzporednem branju podatkov. Nahaja se v predpomnilniku globalnega pomnilnika in njegova trenutna kapacitete je nekje do 64 KB (kilo bajtov). V primerjavi z globalnem ima širšo pasovno širino in kratke dostopne čase. Konstantno spremenljivko deklariramo s ključno besedo `__constant__` in kot navaja njeno ime, jo uporabljamo za shranjevanje preprostih konstantnih vrednosti. Njegova vsebina se prav tako ohrani do konca trajanja programa.

- Teksturni pomnilnik ima podobne lastnosti kot konstantni, le da je bolje prilagojen za dostop do podatkov 2D struktur, kot so tabele in matrike. Podatki v njem so shranjeni v obliki tekstur, ki so njihove normirane vrednosti. Sosednji elementi v strukturah med sabo samodejno interpolirajo. Njegova uporaba zahteva kompleksnejše ukaze v

primerjavi s prejšnjima in ga pri preračunavanju uporabljamo predvsem za shranjevanje večjih struktur podatkov, ki jih med izvajanjem ne spreminjamo.

- Skupni pomnilnik včasih imenujemo tudi vzporedni predpomnilnik. Nahaja se na čipu naprave in gostitelj do njega ne more dostopati. Ta vrsta pomnilnika je dodeljena na ravni samega bloka. Namenjena je izmenjavi podatkov med nitmi v istem bloku in je pri tem veliko hitrejša od globalnega. Njegova trenutna kapaciteta je 16 KB, katero lahko po potrebi razdeli na 16 delov. Pri njegovi uporabi moramo paziti še na sinhronizacijo niti, da ne pride do napak pri preračunavanju. Deklariramo ga s ključno besedo `__shared__`. Vsebina pomnilnika je ohranjena za čas trajanja bloka.

- Zadnja vrsta pomnilnika je registrski pomnilnik. Registri se rezervirajo za vsako posamezno nit, do katerih ima dostop le nit sama. Njihovi dostopni časi so skoraj ničelni.

Poglavje 4

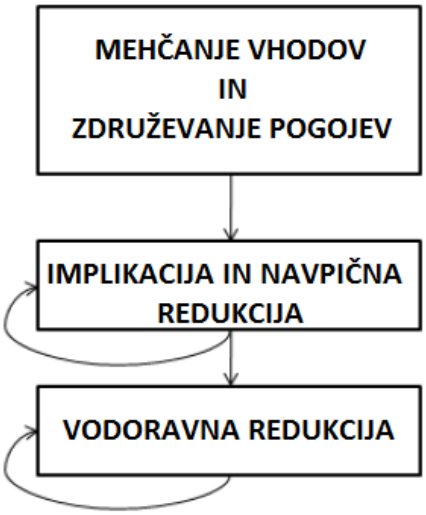
Izvedba rešitve

Ideja za našo izvedbo zahteva, da na naš problem gledamo iz malo drugačnega zornega kota, kot smo vajeni pri rešitvi v zaporedni izvedbi na CPE-ju. Ker delamo v vzporedni izvedbi, moramo operacije poenostaviti, tako da jih razbijemo na večje število enostavnih. Tako bomo namesto procesov preračunavanja funkcij imeli opravka z večjimi tabelami in operacijami `min()` in `max()` med njihovimi vrsticami in stolpci[11]. Celoten postopek implementacije smo razdelili na dva dela: branje in priprava podatkov ter tri jedra, ki opravljajo naloge mehčanja, inference in ostrenja. Slika 2.3.

Prav tako smo se odločili za omejitev sistema na dve vrednosti vhoda in eno vrednost izhoda, da se izognemo kompleksnejši izvedbi, za katero potrebujemo več tabel in operacij med njimi, kar pa ni bistvo tega problema.

4.1 Uporabljena orodja

Problema smo se lotili z nastavitvijo delovnega okolja za razvoj. Za delovno postajo smo uporabili osebni računalnik, ki teče v okolju operacijskega sistema Windows 8. Prednost okolja Windows so predvsem najnovejša orodja in gonilniki, ki jih združba NVIDIA razvija za to okolje. Namestiti smo morali tudi integrirano razvojno okolje Visual Studio Express 12, združbe Microsoft. To je delno omejena verzija enega od najboljših razvojnih okolij



Slika 4.1: Shema procesov rešitve.

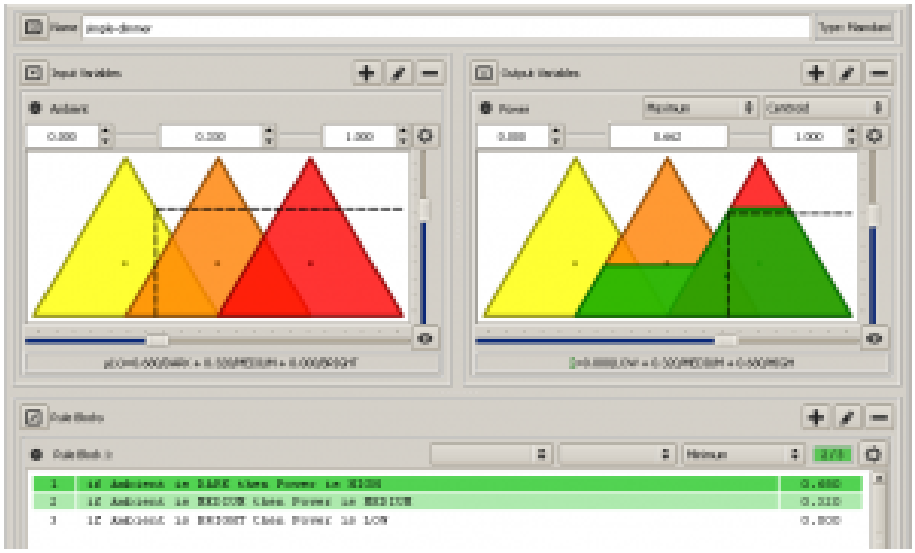
trenutno na tržišču in vsebuje vse, kar je potrebno za namestitev samega paketa orodij CUDA SDK. Uporabljena verzija CUDE SDK je 6.0.

Poleg te izvedbe smo za nadaljnjo razširitev teme uporabili podobno obstoječo izvedbo za CPE iz programskega paketa MATLAB's Fuzzy Logic toolbox, ki smo jo uporabili za primerjavo rezultatov in časov. Kot zadnje orodje smo namestili še programsko knjižico in aplikacijo imenovano fuzzylite, ki smo jo uporabili predvsem za ustvarjanje primerov mehkih sistemov in za vizualizacijo podatkov, kot vidimo na sliki 2.3.

4.2 Branje podatkov in rezervacija prostora na pomnilniku

Po ustvarjenem primeru v orodju fuzzylite lahko ta sistem izvozimo iz aplikacije v različnih formatih datotek, ki se pogosto uporabljajo v podobnih orodjih.

Nas predvsem zanimata formata:



Slika 4.2: Grafični vmesnik aplikacije fuzzylite.

- *.fld* (angl. fuzzylite dataset), ki vsebuje tabelo elementov velikosti m krat n, kjer število vrstic m predstavlja število množic vhodnih in izhodnih podatkov za dani sistem. n po stolpcih pa predstavlja skupno število vseh vhodnih in izhodnih spremenjivk, ki so v našem primeru 3 (2 vhodni in 1 izhodna).

- *.fis* (fuzzy interference system) pa potrebujemo za pridobitev spiska mehkih pravil in parametrov vrednosti pripadnostnih funkcij danega sistema.

Na začetku nas čaka pomemben korak izbire velikosti in tipov pomnilnika za spremenljivke in tabele podatkov, ki jih bomo potrebovali med samimi procesi na napravi in sistemskem pomnilniku.

V sistemskem pomnilniku potrebujemo tabelo za pomnenje vhodnih vrednosti iz datoteke *.fld* in tabelo za spisek pravil, pridobljeno iz *.txt* datoteke. Spisek pravil smo pridobili iz emph.fis datoteke. Potrebujemo še dva vektorja, enega za pomnenje dvojice vhodnih vrednosti in enega za izhodne vrednosti, ki ga z naprave dobimo po zadnjem procesu vodoravne redukcije.

V pomnilniku na globalni ravni potrebujemo tabelo za parametre funkcij pogojnega dela pravil, tabelo diskretiziranih posledičnih vrednosti pravil in tabelo za vmesne rezultate med izvajanjem jeder. Poleg tega potrebujemo še vektor vrednosti združenih pogojev.

Tabela parametrov funkcij pogojnega dela se po pridobitvi podatkov v nadaljevanju programa ne spreminja, zato jo lahko kopiramo v teksturni pomnilnik. Tako nitim omogočimo veliko hitrejši dostop do branja, ki se časovno pozna pri večjih tabelah. Isto lahko storimo s tabelo diskretiziranih posledičnih vrednosti pravil, ki jo po preračunanju pripadnosti po stopnji diskretizacije v nadaljevanju ne spreminjamo več.

Izvorna koda deklaricij, rezervacije prostora tabel in spremenjivk v pomnilnikih:

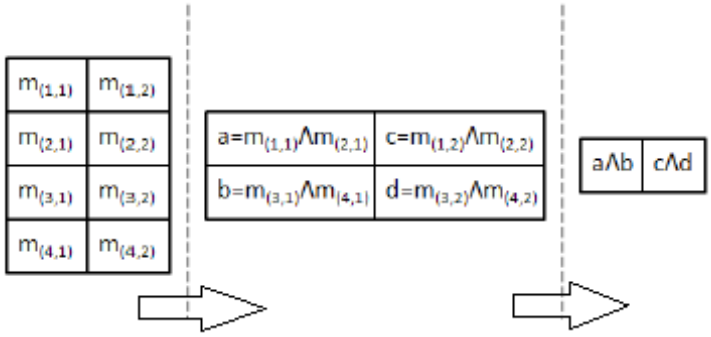
```
//RULE_N = stevilo pravil,
//PARAMETRI_N = stevilo parametrov pripadnih funkcij,
//INPUT_VAL = stevilo dvojic vrednosti vhodov,
//SAMPLE\_RATE = stopnja diskretizacije,
//dvojica vhodnih vrednosti
float FIS_Inputs[2];
//vektor izhodnih vrednosti
float FIS_Outputs[INPUT_VAL];
//parametri funkcij pogojev
float InputRulesFuzzySets[RULE_N][PARAMETRI_N * 2];
//parametri funkcij posledic
float ConsequentFuzzySets[RULE_N][PARAMETRI_N];
//diskretizirane vrednosti posledic
float DiscreteConsequents[RULE_N][SAMPLE_RATE];
float *Results, c;
//tabela shranjevanja rezultatov med scepci
float *CUDA_Implications;
//vektor koncnih rezultatov redukcij
float Implications[SAMPLE_RATE * RULE_N];

//primer rezervacije spomina na napravi
cudaMalloc((void**)&(*Results), RULE_N*sizeof(float));
```

4.3 Proces mehčanja in združevanja pogojev

Tukaj pridemo do prvega vzporednega preračunavanja z jedrom. CPE ovojnica procesa poskrbi za klic ustreznega števila niti. V kolikor imamo manjše število pravil, je to kar enako številu niti, ki se bo pognalo. Tako imamo eno nit za vsako pravilo. V kolikor imamo večje število pravil, pa niti razdelimo še po blokih določenih razsežnosti. Jedro za mehčanje iz teksturnega pomnilnika prebere vrednosti vhodov in parametrov funkcij pogojev. Nato po dani pripadnostni funkciji izračuna vrednosti stopenj pripadnosti obeh vhodov. Na koncu mora še izbrati minimum med obema vrednostima in katerega zapiše v vektor vrednosti združenih pogojev.

Del izvorne kode jedra mehčanja in združevanja:

```
__global__ void Fuzzification(float *dataStream) {
        //Indeksiranje
        unsigned int rule_index = blockIdx.y *
        blockDim.y + threadIdx.y;

        //Preberemo vrednosti iz texturnega pomnilnika
        float x1 = tex2D(FIS_Inputs_tex, 0, 0);
        float x2 = tex2D(FIS_Inputs_tex, 0, 1);

        float a1 = tex2D(InputRulesFuzzySets_tex,
        0, rule_index);
        float b1 = tex2D(InputRulesFuzzySets_tex,
        1, rule_index);

        //...

        //izracunamo stopnje pripadnosti fuzz1, fuzz2
        //po pripadnostni funkciji

        //zdruzimo pogoja pripadnosti z min()
        dataStream[rule_index] = min(fuzz1,fuzz2);
}
```



Slika 4.3: Redukcija po vrsticah.

4.4 Implikacija in navpična redukcija

Redukcija v tem kontekstu pomeni ponavljanje operacije nad vrsto elementov, tako da iz njih dobimo njihov skalarni rezultat[6]. V primeru pravila agregacije je ta operacija maksimum nad stolpci tabele diskretiziranih posledičnih vrednosti. Tukaj v CPE ovojnici kličemo dve skoraj identični jedri navpične redukcije. Prvega kličemo le v primeru, ko je število trenutnih stolpcev večje od maksimalnega števila niti na blok. Njegova funkcija je, da izračuna maksimume med elementi vrstic, kot je prikazano na sliki 4.3. Z izvajanjem jedra zmanjšamo število vrstic za velikost bloka. To jedro še ne izvaja implikacije. Tako zmanjšamo število dostopov do pomnilnika in računskih operacij, ki jih izvedemo v drugem jedru.

Drugo jedro reducira podobno kot prvo in izvede implikacijo nad tabelo diskretiziranih vrednosti. Implikacija je operacija `min()` nad i-to vrstico tabele z i-tem elementu vektorja vrednosti združenih pogojev. Po zadnji redukciji še na koncu preračuna dve vrednosti, ki jih shrani v prvo in drugo vrstico tabele. V prvo vrstico shrani zadnje reducirane vrednosti navpične redukcije, nad katero je bila izvedena implikacija. V drugo vrstico pa shrani produkte reduciranih vrednosti z vrednostmi, ki so indeksi elementov v vrsticah normirani z vrednostjo stopnje diskretizacije (indeks elementa vrstice / stopnja diskretizacije)[12].

Drugo jedro navpične redukcije:

```
__global__ void Last_Step_Reduction_Linear(
             float *dataStream, float *firedAndecents){
        __shared__ float sdata[64];

        unsigned int tid = threadIdx.y;
        unsigned int i = blockIdx.y*blockDim.y +
                threadIdx.y;
        unsigned int j = blockIdx.x;

        //implikacija
        sdata[tid] =
                min(tex2D(DiscreteConsequents_tex,
                        j, i), firedAndecents[i]);

        __syncthreads();

        //redukcija
        unsigned int s;
        for(s = blockDim.y / 2; s > 0; s>>=1){
                if( tid < s ) sdata[tid] =
                        max(sdata[tid], sdata[tid+s]);
                __syncthreads();
        }

        //shranjevanje rezultatov redukcije
        if( tid==0 ){
                dataStream[blockIdx.y + j] =
                ((float)blockIdx.x/(float)SAMPLE_RATE)*
                        sdata[0];
                dataStream[blockIdx.y+SAMPLE_RATE+j] =
                        sdata[0];
        }
}
```

4.5 Vodoravna redukcija

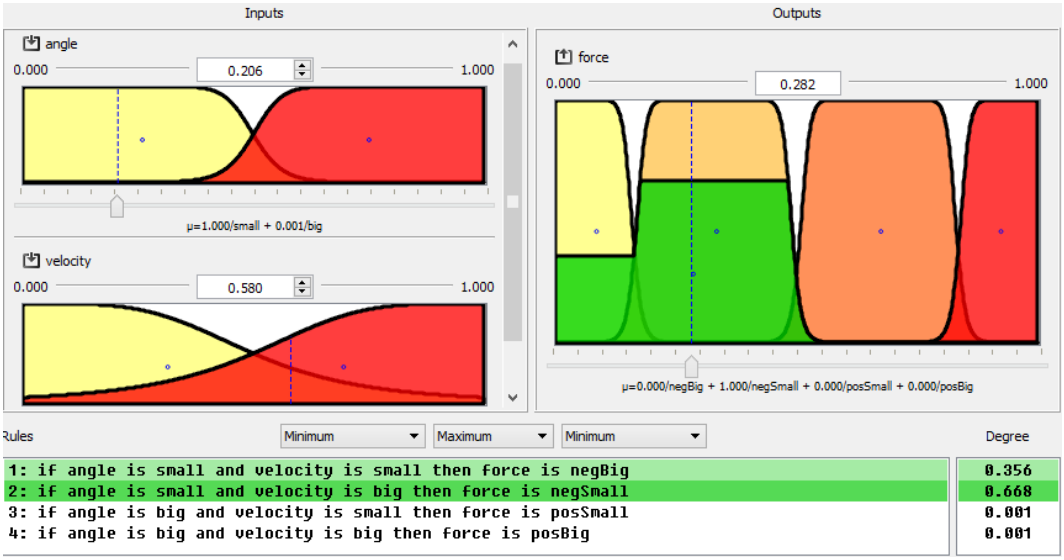
Po prejšnji operaciji nam ostaneta v tabeli dve vrstici, ki ju moramo reducirati še po številu stolpcev. Jedro vodoravne redukcije je skoraj identično jedru navpične redukcije, le da namesto števila vrstic zmanjšuje število stolpcev. Na koncu nam ostane le en stolpec, katerega zgornja vrednost predstavlja števec in spodnja imenovalec funkcije ostrenja s težiščno metodo. Slika 2.3. Na koncu je potrebno le, da se ti vrednosti preneseta nazaj na sistemski pomnilnik, kjer ju delimo. Dobljeni količnik je izhodna vrednost sistema pri dveh danih vhodnih vrednostih.

Poglavje 5

Rezultati in testiranje

Pri testiranju smo si sposodili zaporedno izvedbo mehkega sistema na CPE-ju iz programskega paketa MATLAB's Fuzzy Logic toolbox. Obe izvedbi smo nato primerjali v natančnosti računanja izhodnih vrednosti, v času izvajanja celotnega programa in časih glavnih operacij. Primere mehkega sistema smo pripravili in spreminjali v programskem orodju fuzzylite. Ustvarili smo preprosti primer štirih pravil, dveh vhodov in enega izhoda, prikazanega na sliki 2.3. Za računanje vrednosti pripadnosti smo uporabili generalizirano posplošeno funkcijo zvončaste oblike. Pri uporabi drugih funkcij za računanje pripadnosti se je pokazalo, da so razmerja rezultatov podobna, zato jih v nadaljevanju ne omenjamo. Primeru smo nato dodajali kompleksnost s povečevanjem števila pravil in stopnje diskretizacije. Ti dve vrednosti nekako predstavljata dimenzije glavnih tabel, nad katerimi izvajamo operacije v naši izvedbi. Število pravil smo povečovali tako, da smo obstoječi spisek večkrat prekopirali. Isto smo naredili pri CPE izvedbi, le brez vrednosti stopnje diskretizacije, ki je tukaj konstantna.

Pri obeh izvedbah smo nato iz tabele vhodnih vrednosti izračunali vrednosti izhodov in jih zapisali v zunanjo datoteko. Izvajanje programov smo ponovili 100-krat v zanki, kjer smo dobili povprečje dveh merjenih časov.



Slika 5.1: Uporabljen primer sistema v orodju fuzzylite.

5.1 Natančnost računanja

Natančnost računanja je v naši izvedbi odvisna od vrednosti stopnje diskretizacije, pri večanju katere tudi povečujemo natančnost izhodne vrednosti. Zadostna vrednost te stopnje v našem primeru je okrog 256, kar nam da isto natančnost rezultatov kot CPE izvedba. Pri merjenju časov te vrednosti spreminjamo.

Pri primerjavi rezultatov, zapisanih v zunanjih datotekah, vidimo le manjša odstopanja, ki jih z manjšim zaokroževanjem mantise izenačimo. Pravilnost primerjamo tudi z rezultati orodij fuzzylite in MATLAB, kjer se ujemajo.

5.2 Primerjava časov

Pri merjenju časov nas zanima čas celotnega sistema, ki poleg glavnih procesov zajema še rezervacijo prostora v pomnilniku in kopiranju podatkov. Drugi merjen čas zajema samo čas izvajanja glavnih treh procesov. Spodnji tabeli prikazujeta oba merjena časa v odvisnosti števila pravil in stopnje dis-

pravila/diskret.	128	256	512	CPE
4	302	330	372	40
32	317	356	412	201
128	399	468	593	760
512	551	722	1082	3581

Tabela 5.1: Celotni časi v milisekundah [ms].

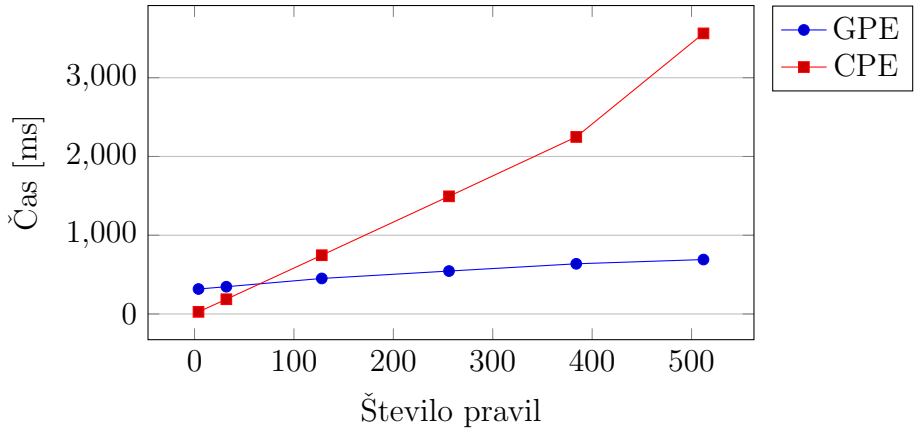
pravila/diskret.	128	256	512	CPE
4	290	317	359	27
32	303	346	398	187
128	384	451	573	746
512	529	691	1038	3563

Tabela 5.2: Časi izvajanja glavnih procesov v milisekundah [ms].

kretizacije. Čase CPE izvedbe vrednotimo z časi GPE izvedbe, s stopnjo diskretizacije 256, kot je prikazano v grafu 5.1. Tabela 5.1 prikazuje dobljene celotne čase, v tabeli 5.2 pa so zapisani časi glavnih procesov. Časovne vrednosti so podane v milisekundah[ms].

Po primerjavi tabel opazimo, da ima CPE izvedba relativno kratke čase pri preračunavanju preprostega sistema z manjšim številom pravil. Celotni časi programa so bistveno podobni časom glavnih operacij, ker imamo opravek samo s sistemskim pomnilnikom. Tako nimamo skoraj nobenih preslikav, hitro dostopne, bralne in zapisovalne čase. Pri večanju števila pravil se veča skoraj samo čas glavnih operacij.

Graf 5.1: Primerjava celotnih časov izvedb.



Pri rezultatih za GPE pa opazimo, da so celotni časi in časi operacij daljši že pri manjšem številu pravil in nizki stopnji diskretizacije. Tukaj vidimo ceno, ki jo plačamo klic in vzpostavitev jedra na napravi. To tudi vključuje rezervacije prostora na različnih pomnilnikih in preslikavah podatkov med njimi. Od te točke dalje se pri dodajanju pravil in višanju stopnje diskretizacije ti časi višajo počasi v primerjavi s CPE. Kot vidimo na Grafu 5.1, v našem primeru GPE prehiti CPE nekje pri sistemu 70-ih pravil.

Poglavje 6

Zaključek

V diplomski nalogi smo prikazali izvedbo mehkega sistema po Mamdanijevem modelu v vzporedni izvedbi na GPE. Rezultati so pokazali, da CPE izvedba prekaša GPE izvedbo pri preprostih sistemih in manjšem številu pravil. Pri dodajanju pravil se časi GPE-ja povečujejo počasi in skoraj linearno, medtem ko se časi CPE-ja začnejo dvigati eksponentno. Sama stopnja diskretizacije se v praksi navadno ne spreminja in njena vrednost je relativno nizka, ker imamo tako najboljše razmerje med natančnostjo in časom preračunavanja.

Poleg tega smo spoznali tudi uporabnost, ki nam jo ponuja arhitektura CUDA. Njena uporaba je preprosta, fleksibilna in razširljiva in jo tako lahko uporabimo skupaj s programskim jezikom C. Naučiti smo se morali le delovanje na ravni, ki nam jo podaja API in spisek potrebnih ukazov.

Po primerjavi vidimo, da je bistvena razlika med izvedbama v hitrosti uporabe pomnilnika in moči procesiranja. Pri procesiranju GPE-ja s povečevanjem števila podatkov komaj opazimo povečevanje procesnih časov. Edina slabost GPE-ja pri našem problemu je rezervacija prostora in zapisovanje v pomnilnik, medtem ko so lahko časi branja hitri in jih lahko bere več niti hkrati. Pri nadaljnji optimizaciji bi tako morali skrbneje načrtovati spremenljivke in strukture v pomnilniku, tako da bi imeli čim krajše dostopne čase in čim manjše število dostopov. Prav tako bi lahko priredili operaciji mehčanja in vodoravne redukcije. Tako da bi sistemu lahko podajali večje

število vhodnih spremenljivk in iz njih dobili več izhodov. Naš program bi lahko tudi pohitrili z boljšo razporeditvijo niti po zvitkih niti in blokih, katerih maksimumi niti bodo v prihodnosti večji.

Literatura

[1] Jernej Virant, "Čas v mehkih sistemih", - Radovljica : Didakta, str. 73-90, 1998.

[2] Nvidia Corp. (2014), "CUDA" Dostopno na: http://www.nvidia.com/object/cuda_home_new.html

[3] Zadeh, L., "Fuzzy Sets", *Information Control* , str. 338-353, 1965.

[4] Shane Cook, "CUDA Programming", *A Developer's Guide to Parallel Computing with GPUs*, 2013.

[5] Jason Sanders, Edward Kandrot, "CUDA by Example", *An Introduction to General-Purpose GPU Programming*, 2010.

[6] Harris, M. (2008), "Optimizing Parallel Reduction in CUDA" Dostopno na: http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[7] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture", *IEEE Micro, vol. 28* , str. 39–55, 2008.

[8] D. B. Kirk, W.-m. W. Hwu, "Programming Massively Parallel Processors: A Hands-on Approach", San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2010.

[9] M. Harris(2009), "Tesla gpu computing." Dostopno na: http://www.cse.unsw.edu.au/ pls/cuda-workshop09/

[10] Nvidia Corp. (2014), "Nvidia cuda c programming guide" Dostopno na: http://developer.download.nvidia.com/compute/

DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[11] Krüger, J., Westermann R. "Linear algebra operators for GPU implementation of numerical algorithms", *Intl. Conf. on Computer Graphics and Interactive Techniques* , str. 908-916, 2003.

[12] D. Anderson, S. Coupland, (2008), "Parallelisation of Fuzzy Inference on a Graphics Processor Unit Using the Compute Unified Device Architecture" Dostopno na: http://http://www.gpucomputing.net/sites/default/files/papers/1206/Parallelisation-of-Fuzzy-Inference-on-a-Graphics-Processor-Unit-Using-the-Compute-Unified-Device-Architecture.pdf

[13] Sanjay Krishnankutty Alonso, (2014), "Mamdani's Fuzzy Inference Method" Dostopno na: http://http://www.dma.fi.upm.es/java/fuzzy/fuzzyinf/main_en.htm