Zachodniopomorski Uniwersytet Technologiczny

w Szczecinie

**WYDZIAŁ INFORMATYKI**

WI_1

Juliusz Romanowski

Kierunek Informatyka

**Zastosowanie technologii CUDA w sztucznej inteligencji**

Praca dyplomowa magisterska

napisana pod kierunkiem

**dr inż. Wiesław Pieruszkiewicz**

w Katedrze Organizacji i Zarządzania

Szczecin, czerwiec 2010

**Oświadczenie**

*Oświadczam niniejszym, że przedkładaną pracę dyplomową kończącą studia podyplomowe napisałem samodzielnie. Oznacza to, że przy pisaniu pracy poza niezbędnymi konsultacjami, nie korzystałem z pomocy innych osób, a w szczególności nie zlecałem opracowania rozprawy innym osobom. Wszystkie dane, istotne myśli pochodzą z literatury i opatrzone są odpowiednim przypisem. Jednocześnie przyjmuję do wiadomości, że gdyby powyższe oświadczenie okazało się nieprawdziwe, decyzja o wydaniu mi świadectwa zostanie cofnięta.*

……………………

własnoręczny podpis

Spis treści

1. Wstęp 6

2. Równoległe przetwarzanie z zastosowaniem GPU 6

3. Sztuczne sieci neuronowe 7

4. Implementacja sztucznych sieci neuronowych na GPU 8

5. Implementacja sztucznych sieci neuronowych na GPU 9

6. Biblioteka CNL 10

A. Zakres programu 10

a) Ogólny projekt aplikacji 10

b) Zastosowania programu. 11

c) Diagramy: 11

d) Struktura plików danych (Opis struktury pliku XML i wczytywanego pliku csv – dane sieci oraz testów). Sekwencja, co się robi przy wczytywaniu i zapisywaniu plikow. 15

i. Plik zestawu testów CSV 16

ii. Plik zestawu testów XML 16

iii. Format Sieci MLP w XML 18

e) Katalogi i pliki w projekcie (Struktura plików i rola każdego pliku) 19

B. Część CPU 20

a) Podejście CPU do aplikacji wykonującej MLP 21

b) Zewnętrzne biblioteki (mersenne twister, TinyXML) 21

C. Część GPU 22

a) Implementacja uruchamiania i trenowania sieci na GPU 22

i. Ogólny algorytm (Inicjalizacja GPU, uruchamianie GPU, różnice miedzy CPU) 22

ii. Struktury danych w pamięci GPU (opisz też ułożenie (kolejność) danych w pamięci GPU). 23

iii. Opis działania kerneli. (opisz wszystkie kernele. opisane, czemu wykonuję kilka testów na raz). 23

iv. Analiza zależności danych. (opis, które dane są zależne. Jakie zrobiłem zabezpieczenia przed racing conditions - konkretnie. ). 23

b) Ograniczenia wykonanego algorytmu. 23

c) Użyte optymalizacje kerneli 23

i. Zrobienie coalesced reads/writes 24

ii. Obsługa 2 testów na raz 24

iii. zapisywanie danych do shared memory. 24

iv. Określenie, ile bloków max. będzie uruchomionych i na tej podstawie deklarowanie ilości pamięci dzielonej. 25

v. Użycie pamięci constant 25

D. Testy implementacji sieci MLP 25

i. Opis danych testowych 25

ii. Porównanie prędkości wykonania operacji na CPU i GPU (execute, train. Porównanie prędkości dla CPU i GPU dla równych struktur sieci. Dyskusja na temat, czy to porównanie jest wiarygodne – weź pod uwagę rodzaje struktur danych na CPU i GPU ; to, że na CPU jest double ; to, że są nowsze CPU i GPU. ) 26

iii. Porównanie działania niezoptymalizowanej i zoptymalizowanej wersji kerneli. 26

iv. Informacje końcowe (Wytłumaczenie, czemu są inne wyniki CPU i GPU - są różnice, bo float zamiast double i inna kolejność obliczeń). 26

E. Narzędzia pomocnicze 26

a) CudaProf (do profilowania) 26

b) Emulacja kerneli na CPU 27

c) Logowanie 27

F. Środowisko testowe 27

a) Hardware użyty w czasie testów 27

b) Porównanie działania w różnych systemach operacyjnych (Windows Vista 32b,Windows 7 64b, Ubuntu 9.04. Porównaj też performance GPU i CPU) 27

7. Wnioski/spostrzeżenia 27

v. Mozliwosc rozwoju programu 28

8. Zakończenie 28

9. Bibliografia 28

10. Załączniki 28

Spis ilustracji

**Nie można odnaleźć pozycji dla spisu ilustracji.**

Spis tabel

**Nie można odnaleźć pozycji dla spisu ilustracji.**

# Wstęp

# Równoległe przetwarzanie z zastosowaniem GPU

GPU są używane do wykonywania programów ogólnego użytku najczęściej, ponieważ oczekuje się od nich większej prędkości niż na CPU

# Sztuczne sieci neuronowe

# Implementacja sztucznych sieci neuronowych na GPU

# Implementacja sztucznych sieci neuronowych na GPU

# Biblioteka CNL

Głównym celem tej pracy magisterskiej jest stworzenie biblioteki obsługującej sieci neuronowe z wykorzystaniem CPU oraz GPU. Biblioteka miała umożliwić dodawanie obsługi różnych typów sieci neuronowych, oraz mieć zaimplementowany jeden z jej typów. Ja zaimplementowałem sieć MLP.

Biblioteka została stworzona w języku C++, zostały stworzone wersje do kompilacji pod Visual Studio 2008 (wersja 32bit i 64bit) oraz G++ (wersja 32bit). Podczas pisania, głównie zwracałem uwagę na prędkość działania części GPU oraz na przenośność programu.

## Zakres programu

Implementacja zawiera bibliotekę razem z plikiem CUDA.cpp, który wykorzystuje tę bibliotekę i w którym są zawarte najczęstsze przypadki użycia biblioteki (m.in. wczytywanie, zapisywanie, uruchamianie sieci na danych testach) .

### Ogólny projekt aplikacji

Obliczenia na GPU są wykonywane w bibliotece CUDA, stworzonej przez firmę nVidia i działającej tylko na kartach graficznych tej firmy, serii 8000 i wyższych. W momencie decyzji o rodzaju biblioteki GPU, na rynku były dostępne 3 biblioteki tego typu: OpenCL, nVidia CUDA, ATI Stream. Poniżej jest tabela wyszczególniająca różne aspekty tych trzech bibliotek.

|  |  |  |  |
| --- | --- | --- | --- |
| Technologia / Aspekt | NVidia CUDA | OpenCL | ATI Stream |
| Obsługiwane chipsety | NVidia GeForce >= 8000, inne nVidia | NVidia GeForce ??? | ATI ??? |
| Popularność | Duża. Popularne forum NVidia | Średnia | Niewielka |
| Częstość uaktualniania biblioteki | Dosyć duża | Średnia | Średnia |
| Prędkość działania aplikacji | Duża | Średnia (w związku z tym, że jest to biblioteka uniwersalna, trudno jest zoptymalizować do konkretnej karty). | Duża |
| Styl programowania | Wysokopoziomowy | Niskopoziomowy, wymaga często więcej linii kodu | Niskopoziomowy, wymaga często więcej linii kodu |
| Różnice w obsłudze równych chipsetów | Tak – CUDA 1.0, 1.1, 1.3, różnice w możliwościach. | ? | ? |

Najważniejszymi aspektami dla mnie przy wyborze biblioteki była łatwość tworzenia kodu, przenośność oraz wsparcie ze strony twórców i osób korzystających z danej biblioteki. W związku z tym wybrałem technologię Nvidia. Uważam, że był to dobry wybór – ani razu nie miałem problemu z rozwiązaniem żadnego problemu z tą biblioteką. Pomagała dokumentacja lub forum NVidia.

W założeniu program miał umożliwić obliczanie wyników sieci tylko za pomocą GPU, ale wolałem dodać możliwość wykonywania tych samych operacji na CPU. Zdecydowałem się na to, ponieważ wg. mnie to jest najlepszy sposób na określenie, czy część GPU działa poprawnie. Inny możliwy sposób sprawdzania poprawności obliczeń na GPU to np. stworzenie sieci o tej samej strukturze w jakimś znanym programie do obsługi sieci neuronowych. Jednak jak dla mnie dużo łatwiejsze debugowanie jest w przypadku, kiedy sam napiszę część CPU. Można wtedy wstawić punkty logowania w podobnych miejscach, co pomaga zauważać błędy.

Innym z założeń było to, żeby biblioteka była łatwo rozszerzalna o obsługę nowych rodzajów sieci neuronowych ( innych niż MLP, którą już zaimplementowałem). Wiele implementacji sieci neuronowych posiada połączone implementacje sieci neuronowej oraz zestawu testów wykonywanego na tej sieci. Ja oddzieliłem te 2 części – dodanie nowego rodzaju sieci nie będzie ingerować w część obsługująćą same testy.

// JRTODO – rozdzielono tez obiekty sieci od obiektow layera i obiektow neuronow. W przypadku obliczen na GPU nie ma obiektu Neuron, ponieważ na GPU dane neuronu są ułożone w specjalny sposób, żeby osiągnąć duza szybkość.

W programie nie ma obsługi warstwy wejścowej sieci. Jej rolę pełni wejście danego testu. Zrobiłem tak, ponieważ wejście testu zawsze ma ten sam typ funkcji aktywacji (funkcja linearna). Oprócz tego nie jest konieczne kopiowanie wartości wejściowych testu do danych warstwy wejściowej.

//JRTODO – napisz, czemu wybralem notacje wegierska

// JRTODO – napisz o generycznosci w sensie tego, ze można zaladowac plik XML sieci dynamicznie nie znajac typu sieci.

// JRTODO – napisz, że proogram przyjmuje wartości wejściowe i wyjściowe jakoliczby zmiennoprzecinkowe. Na CPU to jest double, na GPU float (w zalozeniu moze miec 8b na lepszych GPU).

### Zastosowania programu.

Program umożliwia stworzenie sieci MLP, o dowolnej strukturze. Program obsługuje prostą oraz sigmoidalną funkcję aktywacji. Program obsługuje tylko sieci gęste – w których neuron w warstwie N jest połączony ze wszystkimi neuronami w warstwie N+1. Program obsluguje zadania klasyfikacji oraz optymalizacji (zadania klasyfikacji są obsługiwane przez sprowadzenie do zadania optymalizacji).

// JRTODO – jak zadanie klasyfikacji jest sprowadzane do zadania optymalizacji.

### Diagramy:

Poniżej zamieszczone są diagramy UML biblioteki CNL.

* + - 1. Class



Pliki testów (InputTestSet i InputTest) są niezależne od plików sieci neuronowej.

* + - 1. Sequence



Na powyższym diagramie opisano sekwencję operacji przy uruchomieniu sieci przy użyciu GPU.

Dla każdej warstwy, jest wykonywanych operacji. Najpierw wykonywane są alokacje potrzebnej pamięci dla aktualnej warstwy (wagi, wyjścia) oraz ewentualnie ich zapisanie. Oprócz tego zwalniana jest niepotrzebna pamięć używana wcześniej. Później jest wykonywana operacja executeLayerGPU() – metoda uruchamiająca kernel wykonujący równolegle operacje uruchamiania neuronów na danej warstwie. Sam kernel opisany jest w rozdziale X.X.

* + - 1. Component
      2. Activity

### Struktura plików danych (Opis struktury pliku XML i wczytywanego pliku csv – dane sieci oraz testów). Sekwencja, co się robi przy wczytywaniu i zapisywaniu plikow.

Program, żeby mieć pełnię funkcji, musi mieć możliwość zapisywania i odczytywania stanu pomiędzy uruchomieniami. Poniżej jest zawarta tabela z dostępnymi formatami plików i możliwością ich obsługi:

|  |  |  |  |
| --- | --- | --- | --- |
|  | MLP | InputTestSet | InputTestSet |
| Format | XML | XML | CSV |
| Użycie | Read/Write | Read/Write | Read |
| Zapisane dane | Cała struktura sieci | Wszystkie dane sieci | Dane wszystkich wierszy, bez oznaczenia, które to wyjścia. Czasem są podane nazwy kolumn. |

Obiekt Layer posiada metody saveToXML() i loadFromXML(), przez co jeśli jeśli będzie dodawany jakiś inny rodzaj sieci i będzie korzystał z layerów, to będzie mógł ich użyć.

Poniżej opiszę strukturę podanych typów plików. W poniższej tabeli jest opisana sekwancja do wygenerowania plików XML zestawu testów i sieci MLP.

|  |
| --- |
| InputTestSet testSetCSV; // Nowy zestaw testów  vector<int> vecOutputColumns; // Tworzenie listy numerów kolumn wyjściowych (wynikowych)  vecOutputColumns.push\_back(12); // Jedyna wyjściowa kolumna - indeks 12  vector<int> vecUnusedColumns; // Lista numerów kolumn nieużywanych - pusta  testSetCSV.loadFromCSVFile // Ładowanie listy testów  ("forestfires2.csv" // Plik wejściowy z testami w formacie CSV  ,true // Pierwszy wiersz zawiera nazwy kolumn  ,',' // Określenie znaku oddzielającego elementy - przecinek  ,vecOutputColumns // Podanie listy kolumn wyjściowych  ,vecUnusedColumns); // Podanie listy kolumn nieużywanych  MLP dummyNet; // Nowa sieć MLP  dummyNet.setInputNeuronCount // Ustawienie ilości neuronów wejściowych  (testSetCSV.getInputCount());  dummyNet.addNewLayer // Dodawanie warstwie ukrytej  (6 // Ilość neuronów  ,Neuron::NT\_SIGMOID); // Funkcja aktywancji w warstwie ukrytej  dummyNet.addNewLayer // Ustawienie ilości neuronów wyjściowych ...  (testSetCSV.getOutputCount() // ... - tyle ile wyjść w zestawie testów  ,Neuron::NT\_LINEAR); // Wyjście sieci - linearne  dummyNet.randomizeWeights(0.01,NULL); // dobranie losowych wartości wag  dummyNet.trainNetwork // Uczenie sieci przez CPU  (testSetCSV // Uczenie wcześniej załadowanym zestawem testów  ,6000 // Ilość sekwencji uczenia sieci  ,0.01 // eta - czynnik uczenia  ,1 // Ilość testów uczona na raz  ,NULL); // Generator liczb pseudolosowych - niepotrzebny  dummyNet.executeNetwork(testSetCSV); // Uruchomienie sieci na wszystkich testach przez CPU  dummyNet.executeNetworkGPU(testSetCSV); // Uruchomienie sieci na wszystkich testach przez GPU  testSetCSV.saveToFile("TestSetFromCSV.xml");// Zapisywanie zestawu testów jako XML  dummyNet.saveToFile("NetworkStruct.xml"); // Zapisywanie sieci MLP jako XML |

#### Plik zestawu testów CSV

Plik forestfires2.csv użyty w powyższym kodzie:

|  |
| --- |
| X,Y,month,day,FFMC,DMC,DC,ISI,temp,RH,wind,rain,area  7,5,mar,fri,86.2,26.2,94.3,5.1,8.2,51,6.7,1,0  7,4,oct,tue,90.6,35.4,669.1,6.7,18,33,0.9,0,0  8,4,oct,sat,90.6,43.7,686.9,6.7,14.6,33,1.3,0,1 |

Ten zestaw testów to 3 pierwsze testy z zestawu testów forestfires.csv, dostępnego w Internecie, po drobnej przeróbce. Czasem w plikach CSV pierwsza linia zawiera nazwy kolumn. Jeśli tak jest, należy podać wartość true jako drugi parametr metody InputTestSet::loadFromCSVFile(). Format pliku CSV jest ogólnie znany. Jedyne różnice to znak oddzielający elementy – albo przecinek albo średnik.

Pliki CSV użyte w programie są ogólnodostępne, więc metoda InputTestSet::loadFromCSVFile() zawiera pewne zabezpieczenia przed niepoprawnymi danymi w pliku:

* Minimalnie 3 testy
* Ilość wartości w każdej linii jest taka sama
* Każda z wartości w każdym teście jest niepusta
* Każda kolumna ma przynajmniej 2 wartości

#### Plik zestawu testów XML

Poniżej jest zamieszczony plik TestSetFromCSV.xml wygenerowagany na końcu sekwencji:

|  |
| --- |
| <?xml version="1.0" ?>  <TestSet SourceFileName="forestfires2.csv">  <AttributeMappings>  <AttributeMapping ColumnName="X" IsOutputAttribute="False" ColumnIndexInInputFile="0" ColumnIndexInStructure="0" IsLiteralAttribute="False" MinValue="7" MaxValue="8" />  <AttributeMapping ColumnName="Y" IsOutputAttribute="False" ColumnIndexInInputFile="1" ColumnIndexInStructure="1" IsLiteralAttribute="False" MinValue="4" MaxValue="5" />  <AttributeMapping ColumnName="month" IsOutputAttribute="False" ColumnIndexInInputFile="2" ColumnIndexInStructure="2" IsLiteralAttribute="True">  <ColumnElementName>mar</ColumnElementName>  <ColumnElementName>oct</ColumnElementName>  </AttributeMapping>  <AttributeMapping ColumnName="day" IsOutputAttribute="False" ColumnIndexInInputFile="3" ColumnIndexInStructure="3" IsLiteralAttribute="True">  <ColumnElementName>fri</ColumnElementName>  <ColumnElementName>tue</ColumnElementName>  <ColumnElementName>sat</ColumnElementName>  </AttributeMapping>  <AttributeMapping ColumnName="FFMC" IsOutputAttribute="False" ColumnIndexInInputFile="4" ColumnIndexInStructure="6" IsLiteralAttribute="False" MinValue="86.2" MaxValue="90.6" />  <AttributeMapping ColumnName="DMC" IsOutputAttribute="False" ColumnIndexInInputFile="5" ColumnIndexInStructure="7" IsLiteralAttribute="False" MinValue="26.2" MaxValue="43.7" />  <AttributeMapping ColumnName="DC" IsOutputAttribute="False" ColumnIndexInInputFile="6" ColumnIndexInStructure="8" IsLiteralAttribute="False" MinValue="94.3" MaxValue="686.9" />  <AttributeMapping ColumnName="ISI" IsOutputAttribute="False" ColumnIndexInInputFile="7" ColumnIndexInStructure="9" IsLiteralAttribute="False" MinValue="5.1" MaxValue="6.7" />  <AttributeMapping ColumnName="temp" IsOutputAttribute="False" ColumnIndexInInputFile="8" ColumnIndexInStructure="10" IsLiteralAttribute="False" MinValue="8.2" MaxValue="18" />  <AttributeMapping ColumnName="RH" IsOutputAttribute="False" ColumnIndexInInputFile="9" ColumnIndexInStructure="11" IsLiteralAttribute="False" MinValue="33" MaxValue="51" />  <AttributeMapping ColumnName="wind" IsOutputAttribute="False" ColumnIndexInInputFile="10" ColumnIndexInStructure="12" IsLiteralAttribute="False" MinValue="0.9" MaxValue="6.7" />  <AttributeMapping ColumnName="rain" IsOutputAttribute="False" ColumnIndexInInputFile="11" ColumnIndexInStructure="13" IsLiteralAttribute="False" MinValue="0" MaxValue="1" />  <AttributeMapping ColumnName="area" IsOutputAttribute="True" ColumnIndexInInputFile="12" ColumnIndexInStructure="0" IsLiteralAttribute="False" MinValue="0" MaxValue="1" />  </AttributeMappings>  <Tests>  <Test>  <Inputs>7;5;[-1][mar];[1;-1;-1][fri];86.2;26.2;94.3;5.1;8.2;51;6.7;1</Inputs>  <CorrectOutputs>0</CorrectOutputs>  <NetworkOutputs>-3.52944e-006</NetworkOutputs>  <NetworkOutputsGPU>-3.45707e-006</NetworkOutputsGPU>  </Test>  <Test>  <Inputs>7;4;[1][oct];[-1;1;-1][tue];90.6;35.4;669.1;6.7;18;33;0.9;0</Inputs>  <CorrectOutputs>0</CorrectOutputs>  <NetworkOutputs>1.0557e-005</NetworkOutputs>  <NetworkOutputsGPU>1.06096e-005</NetworkOutputsGPU>  </Test>  <Test>  <Inputs>8;4;[1][oct];[-1;-1;1][sat];90.6;43.7;686.9;6.7;14.6;33;1.3;0</Inputs>  <CorrectOutputs>1</CorrectOutputs>  <NetworkOutputs>0.999976</NetworkOutputs>  <NetworkOutputsGPU>0.999976</NetworkOutputsGPU>  </Test>  </Tests>  </TestSet> |

Jak widać, plik w formacie XML jest zdecydowanie większy od pliku CSV. Jest to spowodowany tym, że:

* Każda wartość jest opisana nazwą
* Podane są również wyjścia sieci obliczone przez CPU i GPU.

// JRTODO – trzeba zauważyć, ze klasyfikacja z 2 możliwymi wartościami to cos innego niż klasyfikacja z 3 i więcej wartościami.

Struktura pliku jest taka, że główny element to TestSet. Zawiera on atrybut SourceFileName, który jest wypełniony jeśli obiekt InputTestSet, który zapisywał dane, był wczytany z pliku CSV. Potomkowie elementu TestSet:

* AttributeMappings – element ten zawiera informacje o wszystkich atrybutach (kolumnach) w danym zestawie testów. Jego potomkami są elementy AttributeMapping (po jednym na każdą kolumną testu). Atrybuty elementu AttributeMapping i ich opisy:
  + ColumnName – nazwa danej kolumny w zestawie testów. Mogła być ustawiona w pliku CSV, który został wczytany, a następnie zapisany do pliku XML.
  + IsOutputAttribute – wartość true/false określająca, czy atrybut jest wyjściowy, czyli czy ma odpowiednik w neuronach wyjściowych.
  + ColumnIndexInInputFile – jeśli zestaw testów pochodzi z pliku CSV, to ten atrybut zawiera indeks kolumny w pliku CSV zawierającej ten atrybut. Jeśli któryś atrybut jest atrybutem klasyfikacyjnym, to ten atrybut oraz ColumnIndexInStructure mogą się różnić.
  + ColumnIndexInStructure – atrybut zawiera indeks danej kolumny w strukturze wewnętrznej programu. Dana kolumna może odpowiadać więcej niż jednej kolumnie w strukturze wewnętrznej programu. Oprócz tego, ta wartość określa indeks danej kolumny w pliku XML w elementach Inputs, CorrectOutputs, NetworkOutputs, NetworkOutputsGPU
  + IsLiteralAttribute – atrybut false/true określający, czy atrybut jest klasyfikacyjny.
  + MinValue – atrybut występuje tylko przy atrybutach liczbowych (IsLiteralAttribute = false). Określa minimalną wartość danej kolumny.
  + MaxValue - atrybut występuje tylko przy atrybutach liczbowych (IsLiteralAttribute = false). Określa mmaksymalną wartość danej kolumny.

Atrybuty literalne (IsLiteralAttribute = false) mają potomków o nazwie ColumnElementName. Każdy z tych potomków zawiera literalną wartość klasyfikacyjną. Jeśli są możliwa tylko dwie wartości, wtedy danemu atrybutowi odpowiada jedno wejście/wyjście (wartość pierwsza odpowiada wartości -1, a druga wartości 1). Jeśli jest więcej wartości, wtedy każda możliwa wartość odpowiada jednemu wejściu/wyjściu. W tym przypadku n-ta wartość powoduje, że wszystkie wejścia/wyjścia przypisane do tego atrybutu mają wartość -1 oprócz n-tego, który ma wartość 1.

* Tests - element ten zawiera informacje o wszystkich testach w danym zestawie testów. Jego potomkami są elementy Test (po jednym na każdy test w zestawie). Elementy potomne elementu AttributeMapping i ich opisy:
  + Inputs – wejścia testu
  + CorrectOutputs – wyjścia wzorcowe
  + NetworkOutputs – wyjścia wygenerowane przez CPU. Nadpisywane przez metodę NeuralNetwork::executeNetwork() lub NeuralNetwork::trainNetwork().
  + NetworkOutputsGPU - wyjścia wygenerowane przez GPU. Nadpisywane przez metodę NeuralNetwork::executeNetworkGPU() lub NeuralNetwork::trainNetworkGPU().

Każdy element w wejściach lub wyjściach jest albo wartością liczbową albo klasyfikacyjną. Są one poprzedzielane średnikami. Wartości liczbowe są przedstawione jako liczby. Wartości klasyfikacyjne są przedstawione liczba/liczby oraz nazwa klasyfikacyjna. Nazwa klasyfikacyjna jest określana po tym, która wartość jest największa. Jeśli k-ta wartość, wtedy „wybrana” została” k-ta wartość klasyfikacyjna.

#### Format Sieci MLP w XML

Poniżej jest plik sieci MLP stworzony po poprzedniej sekwencji operacji w pliku NetworkStruct.xml:

|  |
| --- |
| <?xml version="1.0" ?>  <NeuralNetwork Type="MLP">  <Layer>  <Neuron NeuronType="NT\_SIGMOID">  <Weights>-0.644995;0.0517328;-0.044401;0.0508847;0.553819;-0.640026;-0.0499197;-0.340208;-0.0756364;-0.0542719;0.154846;0.049415;0.0122903;0.0387111;0.0363326;</Weights>  </Neuron>  <Neuron NeuronType="NT\_SIGMOID">  <Weights>-0.492991;0.0634698;-0.0655245;0.0716414;0.426053;-0.491632;-0.0600052;-0.262257;-0.0701903;-0.0706763;0.110501;0.0650343;0.0256953;0.0695995;0.014993;</Weights>  </Neuron>  <Neuron NeuronType="NT\_SIGMOID">  <Weights>-0.597295;0.0589075;-0.0505662;0.0570438;0.509406;-0.600241;-0.0560901;-0.306225;-0.0646669;-0.050627;0.138164;0.0468326;0.0200996;0.0558004;0.0230561;</Weights>  </Neuron>  </Layer>  <Layer>  <Neuron NeuronType="NT\_LINEAR">  <Weights>-1.10815;-0.736504;-0.978474;1.238;</Weights>  </Neuron>  </Layer>  </NeuralNetwork> |

Głównym elementem pliku XML ma nazwę NeuralNetwork. Ma on jeden atrybut Type. Określa on typ sieci neuronowej. Sieć MLP ma w sobie jednego lub więcej potomków Layer reprezentujących warstę ukrytą lub wyjściową. Warstwy są uporządkowane w kolejności od pierszej warstwy ukrytej do warstwy wyjściowej. Ilość neuronów wejściowych jest równa ilości wag w pierwszej warstwie ukrytej (lub wyjściowej, jeśli nie ma warstw ukrytych) minus jeden. Każdy element Layer ma jeden lub więcej potomków Neuron. Element Neuron ma jeden atrybut NeuronType o wartości NT\_LINEAR lub NT\_SIGMOID określający funkcję aktywacji. Element Neuron posiada jednego potomka o nazwie Weights, który zawiera wagi neuronu oddzielone średnikiem.

Jeśli byłby stworzony inny typ sieci neuronowej, mógłby on mieć oczywiście inną strukturę.

Format plików XML MLP i zestawu testów został stworzony tak, żeby łatwiej było dla człowieka to wszystko zrowumieć. Właśnie dlatego został wybrany format XML. Z tego też powodu w pliku zestawu testów znajdują się nadmiarowe dane – w elemencie AttributeMapping jest informacja o minimalneuj i maksymalnej wartości, a można to też wyczytać z konkretnych wartości testów. Jako potomkowie AttributeMapping jest podana lista elementów literalnych, która znajduje się też w potomku Tests. Pliki testów są łatwiejsze do czytania, i dlatego wartości nieliteralne są skalowane do skali -1;1 dopiero po załadowaniu. Zakładamy, że pliki XML są poprawne (bo sam je stworzyłem).

### Katalogi i pliki w projekcie (Struktura plików i rola każdego pliku)

Poniżej znajduje się lista katalogów i plików użytych w projekcie Visual Studio 2008:

|  |  |
| --- | --- |
| **Wstaw FileIndex.txt.**  **Pogrub (albo pochyl) nazwy plików i katalogów** |  |

### Logowanie

W programie dodałem obsługę logowania wiadomości, służy ono do chronologicznej rejestracji różnych zdarzeń. Każda wiadomość ma określony poziom logowania (typ wiadomości). Aplikacja umożliwia dynamiczne określenie, które poziomy logowania są wyświetlane w konsoli, a które w pliku logowania. Jednymi z celów logowania jest określanie sekwencji działań w programie, lub mierzenie długości czasu wykonywania pewnych informacji.

Prawie wszystkie informacje są logowane przy pomocy metody Logging:: logTextFileLine, jednak ograniczenia kompilatora CUDA uniemożliwiają użycie go wewnątrz pliku TrainNetwork.cu – z tego powodu logowanie w tym pliku jest wykonywane przez funkcję printf (tylko do konsoli). Logowanie wewnątrz kerneli jest też ograniczone tylko do trybu emulacji.

W trakcie mojej pracy, miałem pewne problemy z niestabilnym działaniem programu przy wykonywaniu kerneli. Problem ten udało mi się rozwiązać poprzez zapisywanie informacji o adresach i wielkościach zaalokowanych bloków w pamięci GPU, oraz sprawdzanie, czy w którymś momencie nie używałem adresu spoza tablicy. Tryb logowania LT\_DEBUG służy do wyświetlania wielu pomocniczych informacji i również służy to rozwiązaywania błędów, ponieważ w standardowym użytkowaniu programu wypisuje on bardzo dużo informacji.

## Część CPU

W bibliotece, w części CPU, wszystkie dane są zawarte w obiekcie go reprezentującym:

* Obiekt MLP zawiera informacje o warstwach (obiekt Layer) w nim zawartych
* Obiekt Layer zawiera informacje o neuronach w nim zawartych
* Obiekt Neuron zawiera właściwości danego neuronu - typ neuronu, wartości wag, wartości ostatnich wyjść i błędów.
* // JRTODO – zrob to samo dla GPU – gdzie sa tam zawarte dane



Na powyższym diagramie opisano sekwencję operacji przy uruchomieniu sieci przy użyciu CPU. Najpierw są pobierane wejścia testu, później obiekt MLP uruchamia po kolei wszystkie warstwy sieci, a obiekt każdej warstwy oblicza wyjście każdego neuronu w tej warstwie. Później, zapisywane są obliczone wyjścia testu.

// JRTODO – jeszcze TrainNetwork CPU i GPU wklej

### Podejście CPU do aplikacji wykonującej MLP

### Zewnętrzne biblioteki (mersenne twister, TinyXML)

W programie użyłem dwóch różnych bibliotek zewnętrznych – Mersenne Twister i TinyXML.

1. Mersenne Twister

Generator liczb pseudolosowych wbudowany w język c++, rand(), jest bardzo niskiej jakości – działa wolno, a wylosowane liczby nie spełniają wielu testów na losowość. Przez to badania wykonane z tym generatorem mogą nie dawać poprawnych wyników. Dla zadań tej biblioteki, bardzo dobrym rozwiązaniem jest generator Mersenne Twister. Wybrałem bibliotekę obiektową stworzoną przez Richarda J. Wagnera.

1. TinyXML

Jako format pzechowywania danych sieci i zestawów testów, wybrałem XML. Zależało mi na znalezieniu biblioteki do obsługi XML, która przedstawia dokument w postaci struktury obiektów (DOM), oraz jest prosta w obsłudze. TinyXML to jedna z najpopularniejszych darmowych bibliotek dla C++.

## Część GPU

Aktualnie biblioteka umożliwia uruchamianie sieci na danym zestawie testowym oraz uczenie sieci na GPU przy pomocy biblioteki NVIDIA CUDA SDK 2.3. Funkcje (kernele, o nazwach kończących się na ‘kernel’) wykonywane na GPU są zawarte w pliku TrainNetwork.cu. W tym pliku są też zawarte funkcje, które są wykonywane na CPU, a uruchamiaję wykonywaie kerneli (mają nazwy kończące się na CUDA).

Plik CUDATools.cpp zawiera metody statyczne odpowiedzialne za pozostałą komunikację z GPU. Są to operacje alokowania, dealokowania, ustawiania, pobierania pamięci GPU, oraz uruchamiania funkcji z zawartych w TrainNetwork.cu.

### Implementacja uruchamiania i trenowania sieci na GPU

#### Ogólny algorytm (Inicjalizacja GPU, uruchamianie GPU, różnice miedzy CPU)

Wszystkie operacje przy obsłudze biblioteki można podzielić na 3 części:

* Operacje wykonywane an CPU
* Kopiowanie pamięci z GPU do RAM oraz z RAM do GPU.
* Wykonywanie kerneli na GPU

Biorąc pod uwagę taki podział, poniżej zamieszczam diagramy kolejnych operacji (sekwencji):

(dla execute i train)

Na karcie GPU dostępnej dla mnie, nie ma obsługi liczb zmiennoprzecinkowych podwójnej precyzji, więc są użyte liczby pojedynczej precyzji (float). Z tego wzgledu obliczenia na GPU moga byc mniej dokladne.

Sekwencja operacji na GPU dla uruvhamiania i trenowania sieci jest bardzo podobna jak dla CPU. Jednak dla GPU, na początku i końcu operacji, jest konieczne kopiowanie pamięci między RAM a GPU.

#### Struktury danych w pamięci GPU (opisz też ułożenie (kolejność) danych w pamięci GPU).

#### Opis działania kerneli. (opisz wszystkie kernele. opisane, czemu wykonuję kilka testów na raz).

#### Analiza zależności danych. (opis, które dane są zależne. Jakie zrobiłem zabezpieczenia przed racing conditions - konkretnie. ).

### Ograniczenia wykonanego algorytmu.

Jeszcze kilka lat temu jedynymi operacjami możliwymi do wykonania na GPU były operacje bezpośrednio związane z obsługą grafiki. Chociaż NVIDIA umożliwiła wykonywanie innego rodzaju działań na GPU, to cały czas główne zastosowanie pozostaje takie samo – przez co programy ogólnego zastosowania wykonywane na GPU muszą brać pod uwagę pewne ograniczenia.

Ograniczenia mojej biblioteki są właśnie związane z ograniczneniami interfejsu GPGPU . Ograniczenia:

* Każda z warstw może zawierać nie więcej niż 511 neuronów. Jest to związane z maksymalną ilością wątków w bloku.
* Jeden zestaw testów może zawierać max. 65535 testów. Jest to związane z maksymalną ilością bloków w jednym wymiarze gridu.

Możliwe byłoby ominięcie tych ogranicznień, ale byłoby to żmudne, a mogłoby zwolnić działanie programu.

Innego rodzaju ograniczeniem (ograniczeniem samej biblioteki CUDA) jest to, że nie umożliwia wywoływać (klas? Funkcji ze zmienną ilością parametrów?), przez co logowanie wewnątrz kerneli, w trybie emulacyjnym, jest inny niż w reszcie programu.

// JRTODO – optymalizowalem tylko

// JRTODO – ograniczenie, zwykle na GPU jest mniej pamieci niz w RAM.

### Użyte optymalizacje kerneli

Pierwsza wersja pliku XXX.cu zawierała niezoptymalizowane kernele służące do wykonywania i uczenia sieci. Używając programu CudaProf zauważyłem, że przy krótko działających kernelach (napisz ile), czas uruchomienia kernela jest często większy niż jego wykonania. Jest to problemem i powoduje, że krótkie operacje na GPU mogą działać dużo dłużej niż na CPU. Jednym z problemów jest to, że wielkość danych nie jest z góry znana, a w przypadku programów wykonywanych na GPU, trzeba na to bardzo uważać (i trudno przewidzieć jej wyniki) – jedna modyfikacja może w jednym przypadku przyspieczyć działania, a w innym spowolnić. Moje kernele są zwykle zoptymalizowane do przypadków, kiedy mamy dużo danych (duże sieci i wiele testów) i same kernele działają długo.

Poniżej są opisane optymalizacje które użyłem, żeby przyśpieszyć działanie kerneli dla dużych danych.

// JRTODO - dylemat - czy zawsze uzywac iTestIndices przy czytaniu i zapisywaniu, czy tylko na wejsciu pierwszego layera?..

#### Zrobienie coalesced reads/writes

W rozdziale X jest opisane, że w technologii CUDA jest bardzo ważne, żeby odczytywanie i zapisywanie pamięci globalnej było coalesced. Zostało to użyte w następujących przypadkach:

* Wejścia testów, wyjścia testów oraz wyjścia wszystkich layerów są wyrównane (aligned) do 16 elementów. W prawie wszystkich kernelach (oprócz updateWeightsInTrainingKernel),kolejne wątki w bloku reprezenuja kolejne neurony. W zwiazku z czym n-ty wątek czyta/zapisuje n-ty element pamięci, dostęp do pamięci jest coalesced.
* Wagi sieci nie są wyrównane do 16 elementów, ale w kernelach calculateErrorInNotLastLayerKernel i executeLayerKernel te wagi są czytane w wyrównany sposób, tak że najpierw są przepisywane do pamięci dzielonej, a później odczytywane.

#### Obsługa 2 testów na raz

W kernelach executeLayerKernel i calculateErrorInNotLastLayerKernel jeden wątek reprezentuje dwa testy, a w kernelu updateWeightsInTrainingKernel jeden wątek reprezentuje dwie wagi. Ta optymalizacja powoduje, że dane skopiowane w każdym bloku mogą być użyte do wykonania dwóch razy więcej obliczeń. Jednak zmiana prędkości działania kernela executeLayerKernel po tej optymalizacji jest bardzo zależna od wielkości sieci. W przypadku sieci, gdzie 2 kolejne warstwy mają po 500 neuronów, przyspieszenie jest prawie dwukrotne. Przy padku sieci, gdzie 2 kolejne warstwy mają po 20 neuronów, kernel działa ok. 10% wolniej.

#### zapisywanie danych do shared memory.

Pamięć dzielona (shared) jest wiele razy (ile razy) szybsza w dostepie od pamieci globalnej (nawet jak jest coalesced access). Oprócz tego umożliwia łatwe dzielenie się pamięcią między różnymi wątkami wewnątrz bloku. Zmniejszenie operacji bezpośrednio na pamięci globalnej a operowanie na lokalnej jest jest opisywane w poradnikach jako pierwszy krok optymalizacji kerneli. Kernele executeLayerKernel, calculateErrorInNotLastLayerKernel, updateWeightsInTrainingKernel używają pamięci dzielonaj do przechowywania wag, wejść neuronów, błędów na wyjściu neuronu oraz pochodnej wyjścia neuronu. Kernel executeLayerKernel jako jedną z pierwszych operacji, kopiuje globalne dane do tablicy dzielonej:

|  |
| --- |
| // first, we copy d\_LayerInputThisTest to s\_InputNeurons  for(int iInputIndex = threadIdx.x;iInputIndex < p\_iNumInputNeurons; iInputIndex+=blockDim.x)  {  s\_InputNeurons[iInputIndex] = d\_LayerInputThisTest[iInputIndex];  s\_InputNeurons2[iInputIndex] = d\_LayerInputThisTest2[iInputIndex];  PRINT\_MEMORY\_INFO(dp\_pLayerInput,&d\_LayerInputThisTest[iInputIndex]);  }  // we have to make sure that all data was written to shared memory  \_\_syncthreads(); |

Po kopiowaniu danych globalnych do pamięci dzielonej konieczne jest zsynchroniznowanie między wszystkimi wątkami jednego bloku, ponieważ bez tego, możliwa by była sytuacja, gdy jeden warp po wykonaniu pętli, przechodzi do dalszych obliczeń, gdzie czyta dane w pamięci dzielonej, która jeszcze nie była zapisana przez inny warp.

Bardzo podobne kopiowanie odbywa się w dwóch pozostałych kernelach.

#### Określenie, ile bloków max. będzie uruchomionych i na tej podstawie deklarowanie ilości pamięci dzielonej.

Kernel executeLayerKernel używa pamięci dzielonej – jest ona przeznaczana na przechowywanie wejść oraz wag neuronów. Ilość pamięci przeznaczona na przechowywanie wejść jest z góry określona, ale ta na przechowywanie wag – nie. Nie ma gwarancji, że pamięć potrzebna na przechowywanie wag nie przekroczy maksymalnej pamięci dzielonej jednego multiprocesora (16kb). Z tego względu w każdej iteracji pętli, część tablicy wag jest wczytywana do pamięci dzielonej, później przetwarzana, a w kolejnej iteracji pamięć dzielona jest nadpisywana kolejnym fragmentem tablicy wag i przetwarzana. W tym przypadku można zdecydować o wielkości pamięci dzielonej poświęconej na wagi w każdym bloku. Im większa wielkość pamięci dzielonej dla jednego bloku, tym mniej będzie potrzebnych do wykonania iteracji pętli, mniej synchronizacji, ale jednocześnie może być mniejsza ilość bloków równocześnie wykonywanych przez multiprocesor GPU (ponieważ ilość pamięci dzielonej na multiprocesorze wynosi tylko 16kb). W funkcji executeLayerCUDA umieściłem obliczenia optymalnej ilości pamięci dzielonej dla wag. Obliczenia te bazują testach empirycznych, tak żeby kernel działał szybko w różnych przypadkach.

#### Użycie pamięci constant

W przypadku uczenia sieci, wtedy każde wywołanie executeLayerKernel dla pierwszej wartwy wymaga przekazania tablicy indeksów testów, które mają być użyte w danym kroku. Zwykle wielkość tej tablicy nie jest większa niż kilkanaście elementów, ale nie może ona być przekazana jako parametr wywołania kernela – biblioteka CUDA nie pozwala na to. Jednym z rozwiązań byłoby to, że indeksy te byłyby wczytywane za każdym razem z pamięci globalnej. W tym wypadku jednak wymagałoby to użycia pamięci dzielonej, a poza tym każdy blok musiałby kopiować tę samą tablicę globalną. Z tego względu użyłem specjalnego rodzaju pamięci – stałej (constant). Dzięki temu, że ilość pamięci użytej na przechowanie tabeli jest mniejsza niż ilość cache wewnątrz każdego multiprocestora (8kb), dane te będą kopiowane tylko raz do multiprocesora, a później kopiowane z szybkiej pamięci cache.

## Testy implementacji sieci MLP

Jednym z zadań w ramach pisania tej pracy było sprawdzenie działania biblioteki w uczeniu sieci, żeby rozwiązywały rzeczywiste zestawy testów. Dzięki temu:

* Będzie można określić, czy uczenie przez CPU i GPU daje takie same rezultaty
* Czy jakość uczenia jest podobna jak inne znany w na świecie systemy nauki
* Porównać prędkość działania operacji na CPU i GPU.

// napisz, ze na wszelki wypadek robie 3 testy z przypadkowymi wagami

### Opis danych testowych

Wybrałem następujące zestawy testów:

* „Forest Fires” – Baza, gdzie jest 12 wejść określających parametry terenu oraz czasu, a parametrem wyjściowym jest spalony obszar lasu. W większości testów, wartość wyjściowa wynosi 0. Dwa parametry wejściowe (miesiąc i dzień tygodnia) to parametry klasyfikujące.
* “Iris Plants Database” – w Tej bazie są 4 wejścia określające atrybuty irysa. Wyjściem jest jeden atrybut – określający jeden z 3 typów irysów.
* “Wisconsin Diagnostic Breast Cancer (WDBC)” – Baza określająca na podstawie 30 atrybutów wejściowych, czy guz jest łagodny czy złośliwy.

Żaden z tych zestawow testow:

* Nie ma kilku wartości wyjściowych
* Nie ma brakujących wartości

### Porównanie prędkości wykonania operacji na CPU i GPU (execute, train. Porównanie prędkości dla CPU i GPU dla równych struktur sieci. Dyskusja na temat, czy to porównanie jest wiarygodne – weź pod uwagę rodzaje struktur danych na CPU i GPU ; to, że na CPU jest double ; to, że są nowsze CPU i GPU. )

// JRTODO - implementacja CPU uzywa double a GPU floata - wiec daje jej to przewage

// JRTODO – napisz, że według dokumentacji, zaleca się, żeby ilość bloków była przynajmniej równa dwukrotności multiprocesorów w GPU

### Porównanie działania niezoptymalizowanej i zoptymalizowanej wersji kerneli.

Dla kernela execute

### Środowisko testowe

Wszystkie testy zostały przeprowadzone na laptopie z następującą konfiguracją:

* Procesor Intel Core 2 Duo P8400, 2.26Ghz
* 4GB RAM DDR2, 800Mhz
* Karta graficzna z chipsetem NVIDIA GeForce 9600M GT, 512 MB RAM DDR2
* Windows Vista 32bit SP2
* CUDA Toolkit 2.3, CUDA SDK 2.3

Powyzsze/poniższe testy zostały przeprowadzone w środowisku 32 bitowym, jednak jest możliwość uruchomienia go też w środowisku 64 bit. W obu architekturach wszystkie zmienne programu (oprócz wskaźnikowych) mają tę samą szerokość. Program uruchomiłem na Windowsie 7 64bit i uruchomiłem podobne testy. Poniżej znajduje się porównanie prędkości dzialania w obu systemach operacyjnych.

Napisz tez, ze ten procesor jest 64-bitowy Gdyby był 32b, to by było pewnie wolniej

#### Porównanie działania w różnych systemach operacyjnych (Windows Vista 32b,Windows 7 64b, Ubuntu 9.04. Porównaj też performance GPU i CPU)

### Informacje końcowe (Wytłumaczenie, czemu są inne wyniki CPU i GPU - są różnice, bo float zamiast double i inna kolejność obliczeń).

Kod na CPU nie jest zbyt dobrze zoptymalizowany

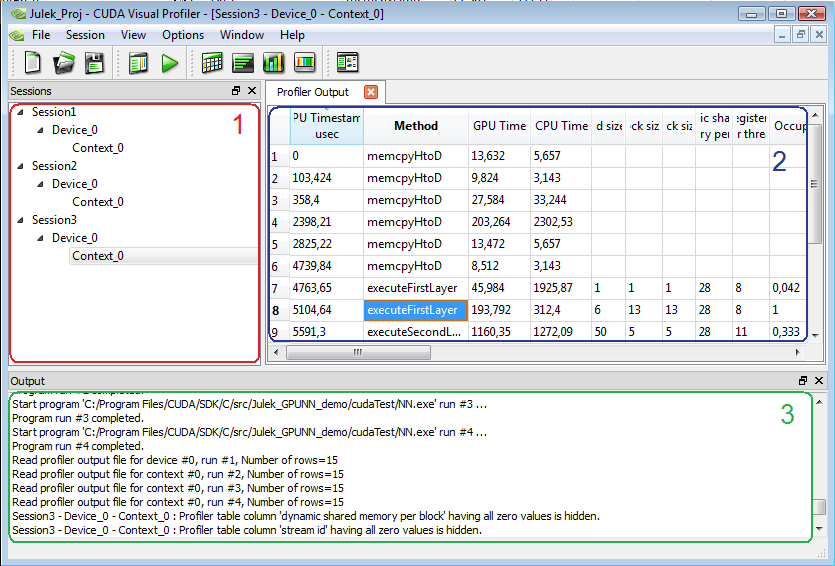
Napisz o najnowszych procesorach i kartach graf. – ile maja razy wiecej mocy niż moje

## Narzędzia pomocnicze

NVIDIA dostarcza razem z biblioteką CUDA aplikację **CudaProf**. Jest to tzw. profiler dynamiczny kerneli CUDA. Program ten uruchamia aplikację CUDA i określa różne parametry kerneli i operacji na pamięci GPU, m.in.:

* Czas wykonania przez CPU, GPU
* Ilość danych zapisów lub odczytanych z pamięci globalnej, z podziałem na coalesced i non-coalesced
* Ilość danych zapisów lub odczytanych z pamięci dzielonej
* Ilość wykonanych instrukcji

Poniżej jest screenshot programu:



Pole 1 przedstawia listę poprzednio uruchomionych symulacji programu. Pole 2 zawiera listę wywołań kerneli oraz operacji na pamięci GPU, razem ze wszystkimi ich właściwościami. Pole 3 jest to okno logowania zawierające pomocnicze dane.

Program udostępnił mi wiele informacji o programie CNL, które były mi bardzo pomocne w procesie optymalizacji kerneli oraz pozwoliły nawet na kilkukrotne przyspieszenie działania części GPU programu.

Inną pomocą w czasie tworzenia programu była możliwość **emulacji** wykonania części GPU na CPU (jest włączana przez specjalny przełącznik kompilacji). Wykonanie całego programu w trym trybie jest możliwe nawet na komputerach nie wyposażonych w karty graficzne wspierające CUDA. Tryb emulacji umożliwia pozwala na debugowanie kerneli (nie jest możliwe w standardowym trybie bez emulacji). Trzeba jednak pamiętać o tym, że jest to emulator, a nie symulator karty graficznej CUDA – występują pewne różnice – niektóre kernele działają poprawnie w trybie emulacji, a w trybie standardowym mogą nawet zawiesić komputer. Kod wykonany w emulatorze jest wielokrotnie wolniejszy niż na GPU.

# Wnioski/spostrzeżenia

// JRTODO - opisz sposob tworzenia programu, problemy, rozwiazania problemow

Żeby napisać program w CUDA, nie jest to trudne. Ale trudne jest, żeby zrobić, żeby działał szybko. Wymagane jest dużo danych, żeby działał szybko. Często inne wersje programu sprawdzają się w jednych sytuacjach, a inne wwersje w innych.

Wartości na GPU są float. Przy niewielu obliczeniach (epokach uczenia, lub warstwach sieci), róznice są niewielkie między double. Przy wielu – stają się coraz większe.

Przyspieszenie przy MLP jest tylko dla duzych sieci.

#### Mozliwosc rozwoju programu

// JRTODO - maybe a possibility to change eta during training?

// JRTODO - zmien mnozenie integerow na specjalna funkcje mnozaca najnizsze 24 bity

Dodanie innych typow NN

Uzycie kilku GPU

MLP bez gestej sieci polaczen

Rozwiazanie problemow z ograniczeniami GPU (max 511 neuronow, itp.)

# Zakończenie

# Bibliografia

# Załączniki