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 Pietruszkiewicz**

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

Abstract

Spis treści

[1. Wstęp 5](#_Toc263329411)

[2. Równoległe przetwarzanie z zastosowaniem GPU 7](#_Toc263329412)

[2.1. Jednostki obliczeniowe GPU 7](#_Toc263329413)

[2.1.1. Różnice między CPU a GPU 7](#_Toc263329414)

[2.1.2. Struktura pamięci GPU 8](#_Toc263329415)

[2.2. Technologie GPGPU 11](#_Toc263329416)

[2.3. CUDA 14](#_Toc263329417)

[2.3.1. Architektura i podstawy CUDA 14](#_Toc263329418)

[2.3.2. Wprowadzenie do technologii 15](#_Toc263329419)

[2.3.3. Proces wykonania programu CUDA 16](#_Toc263329420)

[2.3.4. Wersje CUDA 16](#_Toc263329421)

[2.3.5. Proces kompilacji plików CUDA 18](#_Toc263329422)

[2.3.6. Tryb emulacji 19](#_Toc263329423)

[2.3.7. Model architektury CUDA 19](#_Toc263329424)

[2.3.8. Komunikacja między wątkami 20](#_Toc263329425)

[3. Sztuczne sieci neuronowe 21](#_Toc263329426)

[3.1. Model neuronu 21](#_Toc263329427)

[3.1.1. Neuron biologiczny 21](#_Toc263329428)

[3.1.2. Sztuczny neuron 22](#_Toc263329429)

[3.1.3. Funkcja aktywacji 23](#_Toc263329430)

[3.2. Architektury sieci neuronowych 24](#_Toc263329431)

[3.2.1. Jednokierunkowe 25](#_Toc263329432)

[3.2.2. Sieci radialne 26](#_Toc263329433)

[3.2.3. Sieci Hopfielda 26](#_Toc263329434)

[3.3. Zastosowania sztucznych sieci neuronowych 27](#_Toc263329435)

[3.4. Implementacja sztucznych sieci neuronowych na GPU 28](#_Toc263329436)

[4. Biblioteka CNL 32](#_Toc263329437)

[4.1. Ogólny projekt aplikacji 32](#_Toc263329438)

[4.1.1. Zastosowania programu 34](#_Toc263329439)

[4.1.2. Diagramy 34](#_Toc263329440)

[4.1.3. Struktura plików danych 39](#_Toc263329441)

[4.1.4. Katalogi i pliki w projekcie 40](#_Toc263329442)

[4.1.5. Logowanie 42](#_Toc263329443)

[4.2. Część CPU 42](#_Toc263329444)

[4.2.1. Przebieg procesów na MLP 43](#_Toc263329445)

[4.3. Część GPU 44](#_Toc263329446)

[4.3.1. Przebieg procesów na MLP 45](#_Toc263329447)

[4.3.2. Implementacja MLP na GPU 47](#_Toc263329448)

[4.3.3. Użyte optymalizacje kerneli 49](#_Toc263329449)

[4.3.4. Ograniczenia wykonanego algorytmu 52](#_Toc263329450)

[4.4. Testy implementacji sieci MLP 52](#_Toc263329451)

[4.4.1. Opis danych testowych 53](#_Toc263329452)

[4.4.2. Środowisko testowe 53](#_Toc263329453)

[4.4.3. Wpływ parametrów sieci na jakość uczenia 54](#_Toc263329454)

[4.4.4. Wpływ optymalizacji na wydajność kerneli 56](#_Toc263329455)

[4.4.5. Porównanie wydajności uczenia wersji CPU i GPU 57](#_Toc263329456)

[4.4.6. Informacje końcowe 58](#_Toc263329457)

[4.5. Zewnętrzne biblioteki 59](#_Toc263329458)

[4.6. Narzędzia pomocnicze 59](#_Toc263329459)

[4.7. Możliwości rozwoju programu 60](#_Toc263329460)

[5. Wnioski 62](#_Toc263329461)

[Bibliografia 64](#_Toc263329462)

[Załączniki 69](#_Toc263329463)

[Spis ilustracji 79](#_Toc263329464)

[Spis tabel 80](#_Toc263329465)

# Wstęp

Przez ostatnie kilkadziesiąt lat, wydajność komputerów wzrastała bardzo szybko, w sposób wykładniczy. Według tzw. Prawa Moore’a ilość tranzystorów w procesorach podwaja się co 2 lata. Podobnie jest z mocą obliczeniową komputerów. Sam rozwój sprzętu jest powiązany z rozwojem oprogramowania, które to zużywa coraz więcej zasobów komputera. Firmy produkujące procesory zaczęły coraz bardziej zwiększać ilość rdzeni w CPU, co stworzyło wyzwania przed twórcami oprogramowania – aby całkowicie wykorzystać ich moc, programy muszą dawać możliwość uruchamiania zadań na różnych procesorach lub rdzeniach.

Jednak zaledwie 4 lata temu pojawiła się możliwość wykonywania obliczeń w zupełnie inny sposób. Zamiast używania (tak jak do tej pory) CPU oraz pamięci RAM, pojawiły się biblioteki umożliwiające przetwarzanie danych na procesorach kart graficznych[[1]](#footnote-1), opisane w rozdziale 2. Najnowsze karty graficzne, stworzone do przetwarzania danych graficznych z bardzo dużą prędkością, mogą być również użyte do wykonywania obliczeń ogólnego przeznaczenia. Budowa GPU różni się zdecydowanie od budowy CPU, przez co sposoby wykonywania na nich obliczeń również bardzo się różnią. Na współczesnych GPU jest bardzo dużo jednostek obliczających, a ilość wątków potrzebna do osiągnięcia wysokiej wydajności jest często liczona w tysiącach. W związku z tym, pełne wykorzystanie mocy współczesnych GPU jest możliwe tylko w przypadku zadań, które można w dużym stopniu zrównoleglić.

Do tej pory znaleziono już wiele praktycznych zastosowań GPU w programach ogólnego użytku, jest też wiele prac badawczych porównujących prędkość programów wykonywanych na CPU[[2]](#footnote-2) z programami napisanymi na GPU. Jednym z najczęściej wykonywanych porównań są operacje na sieciach neuronowych. Uruchamianie i trenowanie sieci neuronowych są operacjami, w których wykonywanych jest wiele podobnych działań, przez co są idealne do wykonania na GPU.

Niniejsza praca zawiera opis biblioteki CNL – CUDA Network Library. Jej zadaniem jest wykonywanie operacji na sieciach neuronowych (uruchamianych na CPU oraz GPU), przy użyciu biblioteki NVIDIA CUDA[[3]](#footnote-3). W bibliotece CNL została zaimplementowana obsługa sieci MLP[[4]](#footnote-4); sama biblioteka została zaprojektowana tak, aby była łatwo rozszerzalna o obsługę innych typów sieci. W czasie jej tworzenia, duży nacisk został też położony na uzyskanie wysokiej wydajności operacji na GPU.

Powyższy [rozdział 1](#_Wstęp) zawiera ogólne wprowadzenie do problematyki GPGPU[[5]](#footnote-5), sieci neuronowych oraz opisu stworzonej biblioteki. Opisuje również cele dotyczące niniejszej pracy.

[Rozdział 2](#_Równoległe_przetwarzanie_z) przedstawia informacje o kartach graficznych, ich historię, budowę i technologie służące do użycia ich w celu wykonywania obliczeń ogólnego przeznaczenia. Są tam też wymienione programy i biblioteki wykorzystujące karty graficzne do przyspieszania różnych obliczeń.

Informacje o sieciach neuronowych, ich budowie i rodzajach, są opisane w [rozdziale 3](#_Sztuczne_sieci_neuronowe). Znajduje się tu też opis, w jaki sposób byłoby możliwe wykorzystanie możliwości wielopotokowego procesora GPU przy operacjach na sieciach neuronowych.

Najwięcej miejsca poświęcono na [rozdział 4](#_Biblioteka_CNL), gdzie została dokładnie opisana biblioteka CNL. Zawarte są tam informacje o sposobie implementacji całej biblioteki – części uruchamianej na CPU, jak i na GPU. Opisane zostały sposoby optymalizacji operacji GPU i testy badające prędkość oraz jakość działania biblioteki.

W [rozdziale 5](#_Wnioski) zawarte są wnioski i spostrzeżenia wyciągnięte po napisaniu i przetestowaniu całej biblioteki oraz możliwe perspektywy rozwoju i rozbudowy biblioteki, jak i całego rozwoju technologii GPGPU.

# Równoległe przetwarzanie z zastosowaniem GPU

W rozdziale została przedstawiona historia oraz kolejne udoskonalenia w dziedzinie kart graficznych, szczególnie pod kątem możliwości uruchamiania na nich programów ogólnego zastosowania. Jest w nim też opisana budowa nowoczesnych kart graficznych oraz technologię CUDA.

## Jednostki obliczeniowe GPU

Pierwsze karty graficzne do komputerów PC powstały ok. 30 lat temu i pozwalały tylko na wyświetlanie jednokolorowego tekstu. Niedługo potem pojawiły się ich udoskonalone warianty, które udostępniały możliwość wyświetlania dwuwymiarowego obrazu graficznego, a kolejne wersje pozwalały na coraz wyższą rozdzielczość i ilość kolorów. W 1995 roku pojawiły się pierwsze karty graficzne przetwarzające również tryb 3D[[6]](#footnote-6). W kolejnych latach, napędzane wymaganiami rynku, firmy produkujące chipsety graficzne tworzyły coraz szybsze karty GPU, a wzrost wydajności kart graficznych był zdecydowanie szybszy niż wzrost wydajności CPU[[7]](#footnote-7). Obecnie najszybsze procesory CPU mają wydajność ok. 100 GigaFlops[[8]](#footnote-8), a najszybsze GPU osiągają ponad 1000 GigaFlops, zawierają dziesiątki rdzeni i zapewniają dużą możliwość zrównoleglania operacji.

### Różnice między CPU a GPU

Poniżej zamieszczona jest ilustracja porównująca architektury obu rodzajów procesorów.

|  |
| --- |
|  |
| Rysunek 1. Model fizyczny CPU i GPU  Źródło: (NVIDIA, 2010, str. 3) |

Budowa procesorów graficznych różni się znacząco od budowy CPU. Jest to związane z zadaniami, do których są one wykorzystywane. GPU zawiera wiele jednostek arytmetyczno-logicznych ALU[[9]](#footnote-9) oraz jednostek zmiennoprzecinkowych FPU[[10]](#footnote-10). Jest on dostosowany do rozwiązywania zadań, które mogą być łatwo zrównoleglone (identyczne operacje są wykonywane na dużej ilości danych) – te same instrukcje są używane do przekształceń na wielu tysiącach lub milionach pikseli, tekseli lub wierzchołków. Z drugiej strony, przy przekształceniach graficznych nie jest wymagana zaawansowana kontrola przepływu sterowania[[11]](#footnote-11), a ilość pamięci cache jest ograniczona do minimum. Ilość wątków uruchomionych w danym momencie na GPU jest bardzo duża, jednak narzut czasowy przy ich tworzeniu i kasowaniu jest bardzo mały, w związku z czym programy wykonywane przez każdy z wątków często są krótkie.

### Struktura pamięci GPU

Poniżej na ilustracji pokazany jest model fizycznej struktury procesorów oraz pamięci nowoczesnej karty graficznej firmy NVIDIA:

|  |
| --- |
| http://www.ixbt.com/video3/images/cuda/cuda5.png |
| Rysunek 2. Hierarchia procesorów oraz typów pamięci w karcie graficznej  Źródło: (Berillo, 2008, str. 5) |

Karta graficzna zawiera od kilku do kilkudziesięciu tzw. multiprocesorów[[12]](#footnote-12), pamięć globalną oraz niewielką pamięć constant. W każdym multiprocesorze znajduje się 8 procesorów skalarnych[[13]](#footnote-13), pamięć dzielona, pamięć cache tekstur oraz cache pamięci constant. Pojedynczy procesor skalarny zawiera pewną ilość rejestrów, a jego zadaniem jest wykonywanie operacji arytmetycznych. Każdy z rodzajów pamięci ma specyficzne przeznaczenie:

* Pamięć globalna – ma ona wielkość od kilkuset MB do 4 GB, jest dostępna dla wszystkich wątków; Jej przepustowość wynosi do 100 GB/s, ale opóźnienie przy dostępie wynosi kilkaset cykli zegara, więc zaleca się minimalizowanie używania tej pamięci w programach;
* Pamięć constant – niewielka (64 KB) pamięć zapisywana przed uruchomieniem wszystkich operacji na GPU, zapis do niej nie jest później możliwy; Każdy multiprocesor zawiera 8 KB cache pamięci constant; Ten rodzaj pamięci jest wykorzystywany do przechowywania często używanych danych, które nie są modyfikowane w trakcie działania operacji na GPU;
* Pamięć lokalna – znajduje się w pamięci globalnej, przechowywane są w niej parametry wywołania funkcji GPU lub zmienne lokalne (jeśli brakuje rejestrów);
* Pamięć dzielona – znajduje się na każdym multiprocesorze, ma wielkość od 16 do 48 KB. Może być używana jako cache pamięci globalnej lub do komunikacji między wątkami w danym multiprocesorze;
* Pamięć teksturowa – specjalny rodzaj pamięci mapowany na pamięć globalną; zawiera cache w każdym multiprocesorze;
* Rejestry – W jednym multiprocesorze znajduje się od 8 do 32 tys. 4-bajtowych rejestrów, które są przyporządkowywane do uruchomionych wątków; Są to bardzo szybkie pamięci używane do przechowywania zmiennych lokalnych.

Jak widać, na kartach graficznych jest możliwość wyboru między kilkoma różnymi rodzajami pamięci. Jedną z najważniejszych zasad przy pisaniu programu na GPU jest ograniczenie użycia wolnych typów pamięci do minimum i częste korzystanie z szybszych (pamięć dzielona, cache pamięci teksturowej, cache pamięci constant).

Dla uzyskania wysokiej wydajności przy dostępie do pamięci globalnej i dzielonej, odczyty lub zapisy muszą być wykonywane w odpowiedni sposób – na przykład w technologii CUDA, aby uzyskać maksymalną przepustowość pamięci globalnej, wymagane jest, aby każdy wątek w half-warpie (1 warp zawiera 32 kolejne wątki, half-warp zawiera 16 wątków) dostępował odpowiedniej pozycji tabeli. Na GPU z compute capability*[[14]](#footnote-14)* 1.1, n-ty wątek powinien dostępować pozycji tabeli o indeksie n mod 16. Pozwala to na zapisanie lub odczytanie wszystkich 16 zmiennych za jednym razem (w przeciwnym wypadku te operacje będą wykonane sekwencyjnie). Odczyt lub zapis, w którym za jednym razem przetwarzane są zmienne z ponad jednego wątku, określany jest jako coalesced (złączony). Poniżej jest przykład ilustrujący „złączony” dostęp do pamięci.

|  |
| --- |
|  |
| Rysunek 3. Złączone dostępy do pamięci na urządzeniach CC 1.0 i 1.1  Źródło: (NVIDIA, 2009, str. 84) |

## Technologie GPGPU

GPGPU oznacza użycie GPU do obliczeń do tej pory uruchamianych na CPU(GPGPU.org). Technika ta jest możliwa do zastosowania dzięki kolejnym osiągnięciom w dziedzinie kart graficznych, opisanym poniżej.

Wzrost wydajności kart graficznych w ostatnich latach był powiązany z obsługą coraz większej ilości etapów renderowania przez kartę graficzną. Do czasu stworzenia karty graficznej NVIDIA GeForce 256, etapy transformacji geometrii i obliczania oświetlenia były wykonywane na CPU. Karta ta jako pierwsza umożliwiała obsługę tych operacji[[15]](#footnote-15). Kolejne generacje kart dawały możliwość programowania etapów T&L – specjalne funkcje (tzw. shadery) pozwalały na dodanie np. odbić lustrzanych. Te programowalne etapy pozwalały już na przetwarzanie danych nie-graficznych, jednak nie było to łatwe z kilku względów:

* Języki programowania shaderów (Microsoft HLSL, OpenGL Shading Language) różnią się składnią od popularnych języków programowania;
* Wiele kart nie wspierało obsługi liczb zmiennoprzecinkowych nawet pojedynczej precyzji (32 bity);
* Dopiero późniejsze karty graficzne pozwalały na użycie instrukcji warunkowych i pętli.

Wkrótce potem NVIDIA zaprezentowała platformę CUDA pozwalającą na wykonywanie dowolnych obliczeń na najnowszych kartach graficznych tej firmy. Nie była ona obarczona ograniczeniami języków programowania shaderów, które opisałem powyżej. Firma ATI stworzyła podobny produkt Stream SDK[[16]](#footnote-16) działający na kartach ATI, powstała też biblioteka OpenCL, która wspiera karty graficzne obu tych producentów oraz procesory innych typów. Poniżej umieściłem porównanie tych technologii.

|  |  |  |  |
| --- | --- | --- | --- |
| Tabela . Porównanie technologii GPGPU | | | |
|  | NVIDIA CUDA | OpenCL | ATI Stream |
| Obsługiwane procesory | Karty graficzne NVIDIA, NVIDIA Tesla | Procesory CPU, karty graficzne NVIDIA, ATI, NVIDIA Tesla, procesory DSP i inne (Wilk, 2009, str. 4) | Karty graficzne ATI |
| Popularność, wsparcie produktu | Duża. Popularne forum NVIDIA (NVIDIA) | Średnia | Niewielka |
| Częstość uaktualniania biblioteki | Dosyć duża | Średnia | Średnia |
| Składnia | Podobna do C | Podobna do C | Podobna do C |
| Styl programowania, składnia programów | Wysokopoziomowy lub niskopoziomowy[[17]](#footnote-17) | Wysokopoziomowy | Wysokopoziomowy |
| Źródła: (NVIDIA), (The Khronos Group), (AMD) | | | |

Od czasu powstania frameworków GPGPU, zauważa się coraz większą ilość komercyjnych programów pozwalających na użycie mocy obliczeniowej GPU do wykonywanie niektórych długotrwałych zadań:

* Niektóre projekty w ramach platformy BOINC (np. SETI@home) pozwalają na użycie technologii CUDA lub Stream i dzięki temu kilkukrotnie zwiększają prędkość swojego działania;
* Jest możliwość tworzenia filtrów w programie Adobe Photoshop przy użyciu CUDA;
* Powstał plugin do MATLABa, który wykonuje transformację Fouriera na GPU – dając kilkunastokrotne przyspieszenie;
* PhysX – zestaw narzędzi umożliwiający wykonywanie obliczeń fizycznych na GPU, zaimplementowany w technologii CUDA.

Na stronie internetowej firmy NVIDIA znajduje się pełniejsza lista aplikacji wykorzystujących CUDA. Powstało też wiele prac naukowych opisujących stworzone programy wykonywane na GPU o różnych zastosowaniach, np.:

* rozpoznawanie i klasyfikacja obiektów graficznych,
* zintegrowanie programów Einstein@Home i OpenSteer z NVIDIA CUDA ,
* implementacja algorytmu segmentacji livewire.

W wielu z tych prac osiągnięto wielokrotne przyspieszenie działania w porównaniu z wersjami działającymi tylko na CPU.

## CUDA

Do stworzenia biblioteki obsługującej sieci neuronowe na GPU, autor miał możliwość użycia jednego z trzech podanych wyżej frameworków (NVIDIA CUDA, OpenCL, ATI Stream). Najważniejszymi aspektami przy wyborze jednego z nich była łatwość tworzenia kodu, przenośność międzyplatformowa oraz wsparcie ze strony twórców i osób korzystających z danej biblioteki. W związku wybrana została technologia firmy NVIDIA, która najlepiej spełnia te kryteria.

### Architektura i podstawy CUDA

Aby wykorzystać CUDA, należy posiadać kartę graficzną zgodną z tą technologią, potrzebny jest również specjalny sterownik graficzny oraz pakiet CUDA Runtime. Istnieją też oficjalne biblioteki na tę platformę – CUBLAS (do wykonywania operacji macierzowych) i CUFFT (do wykonywania transformacji Fouriera). Poniżej jest podany schemat elementów składowych:

|  |
| --- |
| http://www.behardware.com/medias/photos_news/00/19/IMG0019322.gif |
| Rysunek 4. Architektura CUDA  Źródło: (Triolet, 2007, str. 4) |

### Wprowadzenie do technologii

Technologia CUDA ma skalowalny model programowania równoległego – program można uruchomić na każdej ilości mikroprocesorów GPU. Jedna funkcja wykonywana na GPU – kernel – opisuje działania wykonywane w jednym wątku. Przy wywołaniu kernela wymagane jest podanie ilości tzw. bloków w gridzie, oraz ilości wątków w bloku. Liczby te określają, ile wątków zostanie uruchomionych równolegle. Zostało to zilustrowane poniżej:

|  |
| --- |
| http://www.behardware.com/medias/photos_news/00/19/IMG0019327.gif |
| Rysunek 5. Podział pracy na bloki i watki  Źródło: (Triolet, 2007, str. 4) |

Każdy blok w gridzie oraz każdy wątek w bloku ma indeks określony maksymalnie trzema wymiarami (na powyższej ilustracji dwoma). Do każdego uruchomionego kernela można użyć innej ilości bloków lub wątków. Po rozpoczęciu kernela, GPU ma za zadanie przypisać każdy blok do wykonania któremuś z dostępnych mikroprocesorów strumieniowych. Z tego względu, aby wykorzystać pełną moc GPU, ilość bloków powinna być przynajmniej równa ilości multiprocesorów w GPU[[18]](#footnote-18).

### Proces wykonania programu CUDA

Większość programów wykonywanych przy pomocy frameworka CUDA, korzysta z niego używając następującej sekwencji działań:

* alokowanie wejściowej i wyjściowej pamięci na karcie graficznej,
* kopiowanie danych wejściowych do pamięci graficznej,
* wykonywanie właściwych operacji (kerneli) na GPU,
* kopiowanie danych wyjściowych z pamięci graficznej do RAM,
* dealokowanie wejściowej i wyjściowej pamięci na karcie graficznej.

Można zauważyć, że jest więcej operacji niż w przypadku wykonywania obliczeń na CPU – jest to związane z kopiowaniem danych między RAM a pamięcią graficzną.

### Wersje CUDA

CUDA może być użyta z kartami GeForce, Quadro lub urządzeniami Tesla (specjalnymi kartami GPU używanymi jako procesory w superkomputerach). Każda z nich posiada określoną architekturę rdzenia (oznaczoną przez tzw. compute capability - CC). Numer wersji compute capability pozwala określić możliwości danej karty graficznej (takich jak ilość pamięci dzielonej lub to, czy dana karta wspiera liczby podwójnej precyzji). Obecne produkty mają CC 1.0, 1.1, 1.2, 1.3 lub 2.0.

Programy CUDA – jako że wykonywane są na GPU, układach które zostały stworzone głównie do wykonywania obliczeń graficznych – mają pewne ograniczenia oraz utrudnienia w porównaniu ze zwykłymi programami zrównoleglonymi na CPU:

* Mnożenie i dzielenie liczb całkowitych jest kilka razy wolniejsze od podobnych operacji na liczbach zmiennoprzecinkowych[[19]](#footnote-19);
* W GPU z compute capability 1.2 lub mniejszym, nie ma obsługi liczb zmiennoprzecinkowych podwójnej precyzji; W procesorach z CC 1.3 są one obsługiwane, ale operacje na nich są kilka razy wolniejsze niż na liczbach pojedynczej precyzji;
* Uruchomienie kernela ma pewien narzut czasowy – przygotowanie kernela może zająć dłużej niż samo jego działanie;
* Aby osiągnąć wysoką wydajność, konieczne jest wykonywanie dużej ilości operacji na GPU proporcjonalnie do ilości danych przesłanych między RAM hosta a pamięcią graficzną (tego rodzaju transfery pamięci są dosyć wolne);
* Jest wiele zasad (zależnych od wersji compute capability GPU) dostępu do pamięci globalnej i dzielonej, które muszą być spełnione, by prędkość działania była optymalna; Zostały one opisane w (NVIDIA, 2010, strony 142-153).

Pierwsze karty wspierające compute capability 2.0 (GeForce GTX 470, GTX 480) zostały wypuszczone na rynek pod koniec marca 2010 roku (Sandhu, 2010). Były one bardzo oczekiwane przez środowisko developerów GPGPU, ponieważ produkty CC 2.0 (tzw. platforma Fermi) miały dawać możliwości niedostępne na poprzednich kartach:

* wykonywanie kilku różnych kerneli równolegle (NVIDIA, 2010),
* obsługa pamięci ECC[[20]](#footnote-20) (NVIDIA, 2010),
* możliwość debugowania wykonania kerneli (NVIDIA, 2010),
* jednolita przestrzeń adresowa wszystkich typów pamięci (NVIDIA, 2009, str. 12),
* możliwość używania klas C++ w kernelach,
* wiele innych drobnych ulepszeń architektury (NVIDIA, 2010, str. 146).

Architektura Fermi usuwa wiele ograniczeń w porównaniu ze starszymi wersjami CUDA i dzięki temu programowanie zrównoleglonych programów wykonywanych na procesorach graficznych może stać się niedużo trudniejsze niż zrównoleglenie programów wykonywanych na CPU.

### Proces kompilacji plików CUDA

Programy CUDA wykonywane na GPU są zapisywane w plikach .cu. Składają się z kerneli, funkcji wywołujących kernele oraz ewentualnie innych funkcji w języku C/C++. Poniżej znajduje się diagram blokowy sekwencji kolejnych przekształceń plików .cu w czasie kompilacji:

|  |
| --- |
|  |
| Rysunek 6. Proces kompilacji plików źródłowych CUDA  Źródło: (Spek, 2008, str. 20) |

Pierwszym etapem przetwarzania jest rozdzielenie przez program cudafe pliku na część wykonywaną na hoście i na część GPU. Następnie część wykonywana na GPU jest kompilowana do postaci binarnej i/lub ptx (odpowiednik assemblera). Przy każdym wywołaniu kernela, odpowiednia postać binarna kernela jest ładowana do pamięci graficznej i uruchamiana na GPU.

Aktualnie są dostępne dwa interfejsy, za pomocą których można pisać programy CUDA: wysokopoziomowy C for CUDA oraz niskopoziomowy CUDA Driver API. Dokładniejsze omówienie oraz przykłady kodu z obu tych interfejsów zostały umieszczone w załączniku 1.

### Tryb emulacji

Pomocą w czasie tworzenia programu jest możliwość emulacji wykonania części GPU na CPU (jest włączana przez specjalny przełącznik kompilacji). Wykonanie całego programu w tym trybie jest możliwe nawet na komputerach niewyposażonych w karty graficzne wspierające CUDA. Tryb emulacji pozwala na debugowanie kerneli (nie jest to 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 przy uruchomieniu na GPU mogą nawet zawiesić komputer. Kod wykonany na emulatorze jest wielokrotnie wolniejszy niż na GPU.

### Model architektury CUDA

Kiedy mikroprocesor dostaje jeden lub więcej bloków do uruchomienia, dzieli on je na specjalne grupy, tzw. warpy (każdy warp zawiera 32 wątki). Wszystkie wątki w warpie w danym momencie mogą wykonywać najwyżej jedną instrukcję. W sytuacji, gdy niektóre wątki warpa wykonują pewną instrukcję warunkową a pozostałe nie, te pozostałe wątki mają zamrożone wykonanie do czasu zakończenia się pętli warunkowej.

Opisana architektura określana jest jako SIMT (ang. Single Instruction, Multiple Threads). Jest ona podobna do architektury SIMD (ang. Single Instruction, Multiple Data) w tym, że dane instrukcje mogą być wykonywane jednocześnie na wielu wątkach. Różnią się one jednak tym, że SIMT pozwala na użycie instrukcji warunkowych, dzięki czemu jest możliwe pisanie kodu dla pojedynczych wątków. Należy jednak pamiętać, że częste używanie instrukcji warunkowych, które rozdzielają ścieżkę wykonania przez wątki w danym warpie znacząco zmniejsza wydajność programu.

### Komunikacja między wątkami

W technologii CUDA szybka pamięć dzielona jest często używana do komunikacji między wątkami w danym bloku (jeden wątek ustawia wartość w tablicy, następnie inny odczytuje ją i interpretuje).

Oprócz tego CUDA udostępnia mechanizm synchronizacji wątków w bloku (jest ona wywoływana przez funkcję \_\_syncthreads). Gwarantuje ona, że dalsze instrukcje w programie będą wykonywane dopiero po dojściu do instrukcji synchronizacji wszystkich wątków w bloku (jest to często konieczne np. w przypadku ładowania danych globalnych do pamięci dzielonej). Nie ma możliwości komunikacji między wątkami w różnych blokach – dlatego należy tak projektować aplikację, by taka komunikacja nie była konieczna.

# Sztuczne sieci neuronowe

Mózg człowieka składa się z ok. 100 miliardów komórek nazywanych neuronami. Każdy neuron ma wiele wypustek łączących go z innymi neuronami lub komórkami efektorowymi (wykonawczymi), a jego działanie polega na generowaniu lub propagowaniu impulsów elektrycznych. Sztuczne sieci neuronowe (ang. Artificial Neural Networks) zostały stworzone jako matematyczny model próbujący symulować biologiczną sieć neuronową. Składają się ze sztucznych neuronów, które na wyjściu mają wartość zależną od wejść w tym neuronie. Sztuczna sieć neuronowa jest systemem adaptującym się, zmieniającym swoją strukturę w czasie fazy uczenia i dopasowującym do danych. Są zwykle używane tam, gdzie standardowy algorytm byłby nieefektywny lub zbyt skomplikowany – w zadaniach takich jak modelowanie skomplikowanych relacji między wejściem a wyjściem, lub wyszukiwanie schematów w danych.

## Model neuronu

Biologiczne oraz sztuczne sieci neuronowe składają się z wielu neuronów, są one podstawą ich budowy.

### Neuron biologiczny

Na poniższym rysunku znajduje się schemat pojedynczego biologicznego neuronu. W neuronie wyróżnia się somę – ciało komórki, oraz dendryty i akson. Dendryty wprowadzają informacje do neuronu, natomiast akson wyprowadza sygnały wyjściowe neuronu. Dendryty kończą się tzw. synapsami, w których sygnał wejściowy może być wzmacniany lub osłabiany. Neurony łączą się przez akson oraz dendryty z wieloma innymi neuronami lub komórkami efektorowymi; liczba tych połączeń może dochodzić do tysiąca.

|  |
| --- |
|  |
| Rysunek 7. Schemat neuronu biologicznego  Źródło: (Wikipedia, 2010), zmodyfikowany |

### Sztuczny neuron

Model sztucznego neuronu jest uproszczeniem neuronu biologicznego. Poniżej jest umieszczona ilustracja symbolizująca działanie sztucznego neuronu.

|  |
| --- |
|  |
| Rysunek 8. Schemat sztucznego neuronu  Źródło: (Klaus), zmodyfikowany |

Wejścia odpowiadają sygnałom nadchodzącym przez dendryty. Wagi to odpowiedniki modyfikacji dokonywanych na sygnałach przez synapsy. Blok sumujący i aktywacji jest odpowiednikiem jądra, a wyjście opowiada aksonowi.

Działanie sztucznego neuronu można opisać następująco: wejścia zawierają wartości, które następnie są mnożone przez odpowiadające współczynniki wag, blok sumujący następnie sumuje te pomnożone sygnały wejść oraz dodaje specjalną wagę – tzw. bias. Zsumowana wartość jest poddana działaniu funkcji aktywacyjnej w bloku aktywacji i podana na wyjście. Wyjście neuronu można więc przedstawić za pomocą następującego wzoru:

|  |  |
| --- | --- |
|  | (3.1) |

Zmienna oznacza -tą wagę, oznacza k-tą wartość wejścia, a  to bias.

### Funkcja aktywacji

Funkcja aktywacji może przybierać różne postacie w zależności od modelu neuronu. Zostaną opisane trzy z nich – neuron liniowy, perceptron Rosenblatta oraz neuron sigmoidalny.

Neuron liniowy nie modyfikuje wyjścia bloku sumującego, jest zwykle stosowany w neuronach warstwy wyjściowej sieci MLP. Wzór jego funkcji aktywacji przedstawia się następująco:

|  |  |
| --- | --- |
|  | (3.2) |

Perceptron Rosenblatta pozwala na klasyfikowanie danych liniowo separowanych. Jego binarna funkcja aktywacji może mieć jedną z dwóch postaci:

|  |  |
| --- | --- |
| lub | (3.3) |

Sieci neuronowe wykorzystujące skokowe funkcje aktywacji (takie jak w perceptronie Rosenblatta) mają cha­rakterystyki nieciągłe, co uniemożliwia zastosowanie niektórych metod uczenia sieci. Wada ta jest wyeliminowana w przypadku neuronów sigmoidalnych. Neuron sigmoidalny ma funkcję aktywacji przybierającą postać sigmoidy. Jest często używany w neuronach warstw ukrytych w sieciach MLP. Wzór funkcji aktywacji tego neuronu może przyjmować dwie postacie:

|  |  |
| --- | --- |
| (funkcja unipolarna)  lub (funkcja bipolarna) | (3.4) |

Parametr w powyższych wzorach określa skos funkcji (najczęściej używana jest wartość 1 ). Wykres funkcji bipolarnej (w zależności od wartości ) wygląda następująco:

|  |
| --- |
|  |
| Rysunek 9. Wykres funkcji y = tanh(ßx) |

## Architektury sieci neuronowych

Pojedynczy neuron mający n wejść, dzieli n-wymiarową przestrzeń na dwie półprzestrzenie oraz może być używany do klasyfikacji obiektów w tej przestrzeni. Może on jednak klasyfikować wektory tylko do jednej z dwóch klas, oraz radzi sobie tylko z problemami liniowo separowalnymi. Aby usunąć te ograniczenia i poszerzyć zakres odwzorowań, należy użyć bardziej skomplikowanych struktur. Poniżej są wymienione trzy z popularnych typów sieci neuronowych.

### Jednokierunkowe

Sieci jednokierunkowe wielowarstwowe (MLP – ang. multilayer perceptron) składają się z odpowiednio połączonych ze sobą neuronów rozmieszczonych w dwóch lub więcej warstwach. W tego typu sieciach, wyjście każdego neuronu w danej warstwie jest przekazywane na wejście każdego z neuronów w warstwie kolejnej. Zwykle oprócz warstwy wejściowej lub wyjściowej występują tzw. warstwy ukryte, które pozwalają na to, by sieć miała możliwość rozwiązywania bardziej skomplikowanych problemów. Poniżej znajduje się schemat sieci jednokierunkowej z warstwą wejściową (neurony oznaczone czarnym kolorem) zawierającą n neuronów, jedną warstwą ukrytą oraz warstwą wyjściową z m neuronami.

|  |
| --- |
|  |
| Rysunek 10. Sieć MLP z jedną warstwą ukrytą |

Sieci jednokierunkowe mogą mieć nieogranicznoną ilość warstw, jednak zwykle korzysta się z sieci z jedną lub dwoma warstwami ukrytymi. Ogólny schemat sieci jednokierunkowej wielowarstwowej wygląda nastęująco:

|  |
| --- |
|  |
| Rysunek 11. Sieć MLP z wieloma warstwami ukrytymi |

Sieć MLP może zostać „nauczona” danego zbioru danych, tak żeby dokładnie lub prawie dokładnie go odwzorowywać. Uczenie wielowarstwowych sieci jednokierunkowych jest przeprowadzane przez wprowadzenie na wejścia sieci wektorów wejściowych oraz modyfikowanie wag, bazując jest „błędzie” wyjścia danego neuronu w porównaniu z oczekiwanym wyjściem. W tym procesie używany jest tzw. algorytm propagacji wstecznej należący do technik uczenia nadzorowanego.

### Sieci radialne

Sieci radialne (RBF – ang. radial basis function networks) służą do interpolacji przestrzeni wielowymiarowej. Neurony w tych sieciach używają funkcji radialnych, których wartość zależy wyłącznie od odległości od określonego punktu (Wikipedia, 2010). Sieci tego typu posiadają trzy warstwy: warstwę wejściową, warstwę neuronów RBF oraz wyjściową (Rutkowski, 2005, str. 220).

### Sieci Hopfielda

Sieci Hopfielda są sieciami rekurencyjnymi (posiadają sprzężenie zwrotne), które mają tylko jedną warstwę neuronów. Wyjścia poszczególnych neuronów są wartościami binarnymi, oraz są podawane na wejścia każdego neuronu. Sieci tego typu są używane jako pamięci autoasocjacyjne.

|  |
| --- |
| http://galaxy.agh.edu.pl/%7Evlsi/AI/hopf/hopfield_pl_pliki/image002.jpg |
| Rysunek 12. Schemat sieci Hopfielda  Źródło: (Kołton & Kwiatkowski, 2005) |

## Zastosowania sztucznych sieci neuronowych

Użyteczność sztucznych sieci neuronowych jest związana z tym, że mogą wyciągać wnioski na podstawie obserwacji. Są używane najczęściej w przypadkach, gdzie złożoność danych jest bardzo duża. Sztuczne sieci neuronowe są zwykle używane do następujących zadań:

* aproksymacja funkcji, regresja, przewidywanie szeregów czasowych, modelowanie,
* klasyfikacja, rozpoznawanie schematów,
* przetwarzanie danych (filtrowanie, klasteryzacja, kompresja),
* robotyka (m.in. sterowanie).

Obecnie najpopularniejsze zastosowania sieci neuronowych to:

* zadania klasyfikacji, rozpoznawania obrazów,
* programy rozpoznawania pisma (OCR),
* przetwarzanie sygnałów,
* prognozy ekonomiczne,
* analiza badań medycznych,
* dobór surowców.

## Implementacja sztucznych sieci neuronowych na GPU

Większość architektur sztucznych sieci neuronowych zakłada ułożenie neuronów w specjalne warstwy. Neurony w tych warstwach wykonują zwykle bardzo podobne działania, co umożliwia wydajne zrównoleglenie operacji wykonywanych na tych neuronach. Zostało stworzonych wiele aplikacji i prowadzono wiele badań na temat użycia sieci neuronowych w klastrach , na superkomputerach lub na dedykowanych architekturach komputerów. Procesory graficzne GPU mogą stanowić tańszą alternatywę dla tych rozwiązań.

Sieć neuronowa może być przedstawiana w kilku postaciach. W modelu obiektowym, każda sieć neuronowa, każda warstwa neuronów oraz pojedynczy neuron są modelowane jako pojedynczy obiekt. W przypadku, gdy zależy nam na najwyższej wydajności, wtedy dane takie jak wejścia, wyjścia testów, wagi, wejścia i wyjścia neuronów mogą być umieszczone w tablicach. Dzięki temu przy operacjach na kolejnych neuronach, odczytywane lub zapisywane są następujące po sobie komórki pamięci, co często zwiększa prędkość działania.

Jako przykład algorytmów używanych w sieciach neuronowych zostaną przedstawione wzory na uruchamianie sieci MLP. Poniżej znajduje się wzór służący do obliczenia wyjścia danego neuronu (lub wyjścia sieci dla danej próbki danych)[[21]](#footnote-21):

|  |  |
| --- | --- |
|  | (3.5) |

, gdzie

|  |  |
| --- | --- |
|  | (3.6) |

W powyższych wzorach, zmienne mają następujące znaczenie:

* oznacza *i*-te wyjście w *k*-tej warstwie sieci,
* oznacza funkcję aktywacji neuronów w *k*-tej warstwie,
* oznacza wyjście bloku sumującego *i*-tego neuronu w *k*-tej warstwie sieci,
* oznacza ilość neuronów w *k*-tej warstwie,
* oznacza *j*-tą wagę *i*-tego neuronu w *k*-tej warstwie sieci,
* oznacza *j*-te wejście w *k*-tej warstwie sieci. Gdy *j*=0, wtedy jest ono równe 1. Gdy *k*=1 (jest to warstwa następna po wejściowej), wtedy ta wartość oznacza *j*-te wejście danego testu. W przeciwnym wypadku, wtedy ta wartość jest równa .

Wagi w *k*-tej warstwie można przedstawić za pomocą następującej macierzy:

|  |  |
| --- | --- |
|  | (3.7) |

Wejścia k-tej warstwy można natomiast przedstawić za pomocą wektora:

|  |  |
| --- | --- |
|  | (3.8) |

Teraz można przedstawić równanie 3.6 w następujący sposób, jako operacja macierzowa (pogrubione zmienne ***x*** i ***t*** oznaczają macierze, oznaczenie identyfikuje i-tą kolumnę w macierzy wag):

|  |  |
| --- | --- |
|  | (3.9) |

Ten wzór można uprościć jeszcze bardziej:

|  |  |
| --- | --- |
|  | (3.10) |

Jak pokazano we wzorze 3.10, obliczanie wyjścia jednej warstwy sieci jest operacją macierzową. Operacje służące do uczenia sieci MLP, opisane w , również mogą być przedstawione jako działania macierzowe. Pozwala to na zrównoleglenie tych operacji.

W przypadku obliczania wyjścia warstwy neuronów, każdemu neuronowi w tej warstwie można przypisać jeden wątek działający równolegle z pozostałymi. W przypadku technologii typu CUDA, taka liczba równoległych wątków może być niewystarczająca dla osiągnięcia wysokiej wydajności. Można jednak tę liczbę zwiększyć przez obliczanie wyjść jednocześnie dla ponad jednego testu. Właśnie tego typu technika jest zastosowana w opisanej w następnym rozdziale bibliotece CNL.

Zrównoleglenie poprzez obliczanie wyjść wielu warstw danej sieci MLP równolegle nie jest możliwe do uzyskania. Powodem jest to, że do obliczenia wyjścia danej warstwy konieczna jest znajomość wyjść warstwy poprzedniej. Podobne ograniczenia występują w procesie uczenia sieci.

Przydatność technologii GPGPU przy użyciu sztucznych sieci neuronowych jest bardzo wyraźna, w związku z czym w ciągu ostatnich kilku lat powzięte były pewne próby wykorzystania tej technologii:

* Praca opisująca użycie technologii CUDA przy wykonaniu kilku różnych algorytmów rozpoznawania obrazów, m.in. sieci SVM[[22]](#footnote-22) – ( (Harvey, 2009), wymieniona wyżej);
* Artykuł opisujący badania nad użyciem splotowych sieci neuronowych (ang. convolutional neural networks) w problemach rozpoznawania twarzy przy użyciu CUDA (Nasse, Thurau, & Fink, 2009);
* Artykuł opisujący użycie technologii Microsoft Accelerator do uczenia sieci MLP;
* Badanie użycia CUDA w symulacji impulsowych sieci neuronowych (ang. spiking neural networks) o dużych rozmiarach .

W powyższych badaniach najczęściej przewijają się wnioski, że użycie procesora graficznego do obliczeń związanych z sieciami neuronowymi może zwiększyć wydajność algorytmu ponad kilkakrotnie. Zwykle jednak, aby osiągnąć takie przyspieszenie, rozmiar problemu i ilość danych muszą być naprawdę duże.

# Biblioteka CNL

Głównym celem tej pracy magisterskiej jest stworzenie biblioteki pozwalającej na wykonywanie operacji na sieci neuronowe z wykorzystaniem CPU oraz GPU. Stworzona biblioteka CNL (CUDA Network Library) obsługuje sieci typu MLP, jednak została zaprojektowana tak, żeby dodawanie innych typów sieci nie było trudne.

Biblioteka została napisana w języku C++, może być kompilowana w środowisku Visual Studio 2008 (wersja 32bit i 64bit) i uruchamiana w systemach operacyjnych Windows XP i wyższych.

## Ogólny projekt aplikacji

Implementacja biblioteki CNL znajduje się w plikach C++ (.cpp, .h) oraz z pliku TrainNetwork.cu (zawiera on implementacje funkcji wykonywanych na GPU). Plik CUDA.cpp nie należy do biblioteki, ale jest wykorzystywany do uruchamiania na niej różnego rodzaju testów. Aktualna implementacja biblioteki pozwala na:

* stworzenie obiektu sieci MLP,
* załadowanie sieci MLP z pliku XML[[23]](#footnote-23),
* zapisanie sieci MLP do pliku XML,
* skonfigurowanie ilości warstw ukrytych, ilości neuronów w tych warstwach i funkcji aktywacji neuronów w każdej warstwie,
* stworzenie obiektu zestawu testów,
* załadowanie zestawu testów z pliku CSV*[[24]](#footnote-24)*,
* załadowanie zestawu testów z pliku XML,
* zapisanie zestawu testów do pliku XML,
* wygenerowanie zestawu testów na podstawie określonej funkcji wyjściowej,
* uruchomienie sieci neuronowej na zestawie testów (na CPU oraz GPU),
* uczenie sieci neuronowej na zestawie testów (na CPU oraz GPU).

W założeniu biblioteka miała umożliwiać obliczanie wyników sieci tylko za pomocą GPU, ale została też dodana możliwość wykonywania tych samych operacji na CPU. Autor zdecydował się na to, ponieważ według niego to jest najlepszy sposób na określenie, czy obliczenia na GPU wykonywane są poprawnie (można to określić przez porównywanie działania sieci uruchomionych na GPU oraz hoście). Innym możliwym sposobem sprawdzania poprawności obliczeń na GPU mogłoby być np. stworzenie sieci o tej samej strukturze w jakimś istniejącym programie do obsługi sieci neuronowych i porównanie jej działania z siecią stworzoną w bibliotece CNL. Nie dawałoby to jednak możliwości porównywania działania algorytmu na GPU i CPU na wszystkich etapach działania algorytmu.

Trenowanie sieci na hoście oraz GPU w bibliotece CNL pozwala na użycie więcej niż jednego testu w każdej iteracji. We większości bibliotek sieci neuronowych jest możliwość użycia tylko jednego testu w jednej iteracji treningu, ale w tej bibliotece zostało to zrobione inaczej ze względu na charakter obliczeń na GPU. Przy jednym teście użytym w każdej iteracji, ilość danych przetwarzanych przed GPU byłaby zawsze niewielka, nawet w przypadku większych sieci neuronowych. Przy większej ilości testów użytych równocześnie, łatwo jest zrównoleglić algorytm na większą ilość użytych bloków.

Jednym z założeń było to, żeby biblioteka była łatwo rozszerzalna o obsługę nowych rodzajów sieci neuronowych (innych niż MLP, którą została już zaimplementowana). Aspekty biblioteki CNL, które ułatwiają dodawanie innych typów sieci:

* Została rodzielona implementacja sieci neuronowej oraz implementacji zestawu testów wykonywanego na tej sieci; Dzięki temu, dodanie nowego rodzaju sieci nie będzie ingerować w część aplikacji obsługującą same testy;
* Istnieją osobne klasy reprezentujące sieć neuronową, warstwę neuronów, oraz pojedynczy neuron, dzięki czemu przy dodawaniu nowego typu sieci będzie możliwe użycie istniejących klas warstwy lub neuronu; Obie te klasy posiadają też zaimplementowaną możliwość zapisywania lub odczytywania właściwości do/z pliku XML;
* Możliwe jest załadowanie danych sieci neuronowej z pliku XML bez znajomości typu tej sieci;

W bibliotece, warstwa wejściowa sieci nie ma odpowiadającego jej obiektu Layer. Jej rolę pełni wejście danego testu. Zostało to tak zrobione, ponieważ neurony wejściowe zawsze mają ten sam typ funkcji aktywacji (funkcję linearną). Dzięki temu nie jest też konieczne kopiowanie wartości wejściowych testu do obiektów warstwy wejściowej.

Po stronie hosta, wszystkie zmienne są przechowywane jako liczby zmiennoprzecinkowe podwójnej precyzji (double), jednak w związku z ograniczeniami technologii CUDA[[25]](#footnote-25), na komputerze autora operacje po stronie GPU są przetwarzane z wykorzystaniem liczb pojedynczej precyzji (float). Gdyby biblioteka była użyta na karcie graficznej wspierającej compute capability 1.3 lub wyższą, możliwe by było użycie liczb podwójnej precyzji[[26]](#footnote-26).

### Zastosowania programu

Biblioteka CNL służy do rozwiązywania zadań aproksymacji lub klasyfikacji. Każdy rekord w zestawie testów może mieć wiele wejść oraz wyjść. Każdy atrybut może być liczbowym atrybutem z pewnego zakresu, lub atrybutem symbolicznym (klasyfikacyjnym). Każdy atrybut symboliczny zostaje sprowadzony do jednego lub więcej atrybutów liczbowych (w szczególności – zadanie klasyfikacji jest sprowadzane w programie do zadania aproksymacji).

W bibliotece jest możliwość stworzenia sieci MLP, o dowolnej strukturze. Program obsługuje prostą funkcję aktywacji oraz sigmoidalną funkcję aktywacji w postaci unipolarnej oraz bipolarnej. Wspierane są tylko sieci gęste – w których każdy neuron w warstwie N jest połączony ze wszystkimi neuronami w warstwie N+1.

### Diagramy

W tym rozdziale zamieszczone są diagramy UML[[27]](#footnote-27) opisujące bibliotekę CNL. Znajduje się tu diagram komponentów biblioteki oraz powiązanych z nią elementów, dwa diagramy klas, oraz diagram sekwencji uczenia sieci. Oprócz tego, w rozdziałach 4.2.1 i 4.3.1 są szczegółowe diagramy sekwencji uruchamiania i uczenia sieci. Wszystkie diagramy zostały stworzone w programie Visual Paradigm for UML 7.2.

Na poniższym diagramie widać, że program używający biblioteki, zarządza osobno obiektami zestawu testów i sieci neuronowej. Obiekt sieci neuronowej (np. obiekt klasy MLP) wywołuje metody statyczne klasy CUDATools, które z kolei wywołują kernele. Biblioteki zawarte w CUDA Toolkit oraz sterowniku graficznym kopiują wersję binarną kerneli do pamięci graficznej i uruchamiają je.

|  |
| --- |
|  |
| Rysunek 13. Diagram komponentów biblioteki CNL oraz jej otoczenia |

Poniżej znajdują się dwa diagramy. Pierwszy z nich zawiera klasy modelujące sieci neuronowe oraz związane z nimi obiekty. Na drugim są zobrazowane pozostałe klasy – związane z testami oraz zestawami testów. Diagramy te zostały wygenerowane przez program UML na podstawie kodu, dzięki czemu te diagramy dokładnie opisują kod aplikacji.

|  |
| --- |
|  |
| Rysunek 14. Diagram klas biblioteki CNL (część pierwsza) |

|  |
| --- |
|  |
| Rysunek 15. Diagram klas biblioteki CNL (część druga) |

Poniższy diagram ilustruje przypadek użycia biblioteki – uczenie nowo stworzonej sieci MLP. W tym celu najpierw należy stworzyć obiekt zestawu testów (InputTestSet) oraz załadować testy z pliku. Następnie trzeba stworzyć obiekt sieci (klasy MLP), skonfigurować jego strukturę, a następnie wykonać procedurę uczenia (wykonaną na hoście lub na GPU). Po zakończeniu uczenia, można uruchomić sieć na podanym zestawie testów i zapisać zaktualizowany plik sieci neuronowej i plik zestawu testów. Kod odpowiadający razem z dokładnym opisem wszystkich parametrów metod znajduje się w rozdziale 4.1.3.

|  |
| --- |
|  |
| Rysunek 16. Diagram sekwencji uczenia sieci w bibliotece CNL |

### Struktura plików danych

Program wykonujący operacje na sieciach neuronowych, żeby mieć pełnię funkcjonalności, musi mieć możliwość zapisywania i odczytywania stanu pomiędzy uruchomieniami. Poniżej jest zawarta tabela z dostępnymi formatami plików i możliwością ich obsługi:

|  |  |  |  |
| --- | --- | --- | --- |
| Tabela . Formaty plików danych obsługiwane przez bibliotekę CNL | | | |
|  | MLP | InputTestSet | InputTestSet |
| Format | XML | XML | CSV |
| Użycie | Odczyt/Zapis | Odczyt/Zapis | Odczyt |
| Zapisane dane | Cała struktura sieci | Wszystkie dane zestawu testów | Wszystkie dane zestawu testów oprócz wyjść sieci dla podanych testów |

Ogólnodostępne zestawy testów są zwykle zapisane w formacie CSV, więc została dodana możliwość ich odczytania. Autor chciał też dodać przydatną możliwość wczytywania i zapisywania stanu sieci neuronowej lub zestawu testów. Zależało mu na tym, żeby format plików był czytelny również dla użytkownika biblioteki i zawierał wszystkie informacje o stanie danego obiektu. W związku z tym został wybrany popularny format plików XML. Dokładny format trzech powyżej opisanych typów plików znajduje się w załączniku 2.

W poniższej tabeli jest opisana sekwencja generująca pliki 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 aktywacji 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 |

### Katalogi i pliki w projekcie

Poniżej znajduje się lista katalogów i plików użytych w projekcie Visual Studio 200. Na czerwono zaznaczone są katalogi programu, natomiast na brązowo z wcięciem są zaznaczone pliki. Jeśli dana klasa jest umieszczona w pliku .cpp oraz .h, to została wymieniona jako NazwaPliku.\*.

|  |  |
| --- | --- |
| Tabela . Lista katalogów i plików w projekcie razem z opisami | |
| Nazwa katalogu/pliku | Opis |
| CUDAFiles | Znajdują się tu wszystkie pliki mające bezpośredni związek z CUDA (pliki .cu, pliki alokujące, kopiujące, usuwające pamięć GPU). |
| CUDATools.\* | Klasa służąca do alokowania, kopiowania, usuwania pamięci GPU oraz wywoływania funkcji z pliku TrainNetwork.cu. |
| TrainNetwork.cu | Metody wywołujące kernele oraz kernele obliczający wyjście warstwy sieci i kernele uczące sieć. |
| Global | Pliki potrzebne do ogólnego działania projektu. |
| CUDA.cpp | Zawiera funkcję main programu, wywołuje podstawowe operacje na sieci i testach, sprawdza poprawność działania programu. |
| Global.\* | Globalne funkcje, nie pasujące nigdzie indziej. |
| Logging.\* | Klasa logująca. Zapisuje logi do pliku i/lub na ekran. |
| MersenneTwister.h | Klasa generująca liczby pseudolosowe. Bliżej opisana w rozdziale 4.5. |
| stdafx.\* | Prekompilowany nagłówek – użyty dla szybszej kompilacji. |
| InputTests | Katalog zawierający klasy InputTest i InputTestSet. Te klasy służą do zarządzania testami, w założeniu mogą być używane też w innych rodzajach sieci niż MLP. |
| AttributeMapping.\* | Klasa zawiera informacje o mapowaniu jednego atrybutu (kolumny) zestawu testów. |
| InputTest.\* | Zawiera informacje o jednym teście - wejścia, oczekiwane wyjścia, oraz ostatnie wyjścia sieci wygenerowane na CPU i GPU. |
| InputTestSet.\* | Zawiera informacje o zestawie testów. Zestawy testów można wczytywać/zapisywać do pliku. |
| MLP | Pliki związane bezpośrednio z implementacją sieci neuronowej MLP. |
| MLP.\* | Zawiera informacje o sieci MLP. Klasa dziedziczy z klasy NeuralNetwork. Implementuje wiele metod abstrakcyjnych z klasy bazowej. |
| NeuralNetwork | Pliki związane z ogólnym działaniem sieci neuronowych. W założeniu mają z nich korzystać wszystkie rodzaje sieci neuronowych. |
| Neuron.\* | Klasa zawiera informacje o generycznym neuronie. |
| Layer.\* | Klasa zawiera informacje o wartstwie sieci. |
| NeuralNetwork.\* | Klasa bazowa sieci neuronowej. Zawiera wiele metod abstrakcyjnych do zaimplementowania w klasach pochodnych. |
| Resources | Pliki nie używane podczas kompilacji. |
| Cuda.Rules | Plik zawierający informacje o kompilacji plików .cu dla Visual Studio 2008. |
| Diagrams\_Visual\_Paradigm.vpp | Diagramy UML w programie Visual Paradigm. |
| Wykres\_JakoscUczenia.xlsx | Pomocniczy plik Excela z wygenerowanymi wykresami jakości uczenia. |
| TinyXML | Wszystkie pliki związane z biblioteką TinyXML (w niewielkim stopniu zostały zmodyfikowane pliki tinystr.cpp i tinystr.h). |
| tinystr.\* | Ta klasa została zmodyfikowana, aby zawierała przydatne dodatki - metodę format (podobna do sprintf) oraz konstruktor akceptujący podobne parametry jak metoda format. |
| \*.\* | Pozostałe pliki z biblioteki TinyXML. |

### Logowanie

W programie została dodana obsługa logowania wiadomości do pliku lub na konsolę, które służy do chronologicznej rejestracji różnych zdarzeń. Każda wiadomość ma określony „poziom” logowania (typ wiadomości), np. LT\_INFORMATION (informacje o przebiegu programu), LT\_WARNING (ostrzeżenia o potencjalnych błędach), lub LT\_ERROR (błędy lub niespełnione warunki asercji). Aplikacja umożliwia dynamiczne określenie, które poziomy logowania są wyświetlane w konsoli, a które w pliku logowania. Logowanie umożliwia:

* łatwe określenie sekwencji działań wykonywanych w bibliotece,
* mierzenie czasu wykonywania pewnych operacji,
* kilka razy w czasie pisania programu, logowanie pewnych danych bardzo ułatwiło autorowi rozwiązanie problemów z niestabilnym działaniem biblioteki,
* logowanie pewnych informacji o wszystkich kolumnach testów w czasie wczytywania zestawu testów z pliku CSV,
* wyświetlanie informacji, jak duże są różnice między poprawnymi wyjściami testów a wyjściami sieci neuronowej (metoda InputTestSet::printVectorDifferenceInfo).

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.

## Część CPU

W bibliotece została zaimplementowana możliwość uruchamiania i uczenia sieci MLP wykonywana bez użycia GPU. Każdy obiekt sieci MLP (czyli sieć neuronowa, warstwa neuronów oraz pojedynczy neuron) posiada reprezentującą go klasę (odpowiednio klasy MLP, Layer, Neuron). Każdy obiekt jednej z tych klas zawiera następujące informacje:

* Obiekt MLP zawiera informacje o warstwach (obiekty 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.

### Przebieg procesów na MLP

Poniżej są zamieszczone dwa diagramy przedstawiające uruchamianie i trenowanie sieci MLP razem z opisami. Na pierwszym z nich opisano sekwencję kolejnych wywołań metod 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.

|  |
| --- |
|  |
| Rysunek 17. Diagram sekwencji uruchamiania sieci na CPU |

Na kolejnym diagramie opisano sekwencję operacji dla jednej iteracji trenowania sieci MLP przy użyciu CPU. Pierwszym etapem jest wybranie losowych testów z zestawu testów oraz uruchomienie ich. Następnie określany jest błąd wyjściowy każdego neuronu. Końcowym etapem jest zaktualizowanie wszystkich wag.

|  |
| --- |
|  |
| Rysunek 18. Diagram sekwencji trenowania sieci na CPU |

## Część GPU

Do zaimplementowania części biblioteki uruchamianej na GPU, użyty został interfejsu C for CUDA[[28]](#footnote-28). Wszystkie kernele[[29]](#footnote-29) oraz funkcje języka C służące do wywoływania kerneli, zawarte są w pliku TrainNetwork.cu. Kernele są oznaczone przez słowo kluczowe \_\_global\_\_, a ich nazwy kończą się słowem „Kernel”, natomiast funkcje wywołujące kernele mają nazwy z zakończeniem „CUDA”. Pozostałe funkcje używające wywołań CUDA znajdują się w pliku CUDATools.cpp. Zawarte są tam 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.

### Przebieg procesów na MLP

Na diagramie poniżej opisano sekwencję operacji przy uruchomieniu sieci przy użyciu GPU. Dla każdej warstwy jest wykonywanych kilka operacji. Najpierw alokowana jest pamięć potrzebna dla aktualnej warstwy (wagi, wyjścia) i zwalniana jest niepotrzebna pamięć używana wcześniej. Następnie jest wykonywana metoda statyczna executeLayerGPU, która uruchamia kernel wykonujący równolegle operacje uruchamiania neuronów na danej warstwie. Sam kernel opisany jest dokładniej w rozdziale 4.3.1.5.

|  |
| --- |
|  |
| Rysunek 19. Diagram sekwencji uruchamiania sieci na GPU |

Na kolejnym diagramie opisano sekwencję operacji dla jednej iteracji uczenia sieci MLP przy użyciu GPU. Pierwszym etapem jest wybranie losowych testów z zestawu testów oraz uruchomienie ich. Następnie określany jest „błąd” wyjściowy każdego neuronu. Na końcu każdej iteracji wszystkie wagi sieci są aktualizowane.

|  |
| --- |
|  |
| Rysunek 20. Diagram sekwencji trenowania sieci na GPU |

### Implementacja MLP na GPU

Algorytmy uruchamiania i trenowania sieci przez wsteczną propagację błędu wykonywane na CPU i GPU są identyczne (różnice występują jedynie w kolejności dodawania kolejnych czynników). Mimo to, wykonanie tych algorytmów na procesorze graficznym jest bardziej skomplikowane ze względu na konieczność kopiowania danych z RAM do tablic w pamięci graficznej i z powrotem oraz równoległy charakter operacji na GPU.

Technologia CUDA pozwala na użycie złożonych typów danych (struct), jednak żeby uzyskać wysoką wydajność, konieczne jest czytanie oraz zapisywanie danych z/do globalnej pamięci przy pomocy tablic[[30]](#footnote-30). Z tego względu, po stronie GPU używane są tylko zmienne całkowite, zmienne zmiennoprzecinkowe oraz tablice jednowymiarowe zmiennych tych typów. Alokowanie i dealokowanie pamięci na GPU zajmuje dużo więcej czasu niż w pamięci RAM, więc nie ma osobnych tablic dla pojedynczego neuronu lub pojedynczego testu – tablice zawierają informacje o całej warstwie neuronów lub zestawie testów. Poniżej znajduje się lista tablic używanych po stronie GPU oraz ich opis:

* TestsInput – wartości wejść testów,
* TestsOutput – wartości wyjść testów,
* LayerWeights – wagi w danej warstwie neuronów,
* DerivativeOfLastOutput – pochodna wyjść neuronów w danej warstwie,
* LastOutputWithOutputFunction – wyjścia neuronów po wykonaniu funkcji aktywacji,
* LastError – różnica między oczekiwanymi a rzeczywistymi wyjściami neuronów,
* TestIndices – indeksy testów używanych przy trenowaniu sieci; znajduje się w pamięci stałej (constant).

Oprócz wyżej wymienionych tablic, kernele używają tablic znajdujących się w pamięci dzielonej, która jest używana jako szybka pamięć podręczna.

Wszystkie operacje w technologii CUDA wykonywane na GPU znajdują się w kernelach (odpowiednikach funkcji wykonywanych na CPU). Poniżej są wymienione i opisane kernele użyte w bibliotece.

|  |  |  |
| --- | --- | --- |
| Tabela . Opis użytych kerneli | | |
|  | executeLayerKernel | calculateErrorInLastLayerKernel |
| Opis | Oblicza wyjście neuronów w danej warstwie na podstawie wejść i wag neuronów. | Oblicza błąd w ostatniej warstwie sieci. |
| Użycie | Uruchamianie i trenowanie sieci. | Trenowanie sieci |
| Ilość bloków | W przypadku uruchamiania sieci - liczba testów w zestawie testów;  W przypadku trenowania sieci – ilość uczonych elementów; | Ilość uczonych elementów |
| Ilość wątków | Ilość neuronów w danej warstwie | Ilość wyjściowych neuronów |
| Użycie shared memory | Wejścia warstwy i wagi neuronów | Nie |
|  | calculateErrorInNotLastLayerKernel | updateWeightsInTrainingKernel |
| Opis | Oblicza i zapisuje błąd w warstwie sieci innej niż ostatnia. | Uaktualnia wartości wag w danej warstwie. |
| Użycie | Trenowanie | Trenowanie |
| Ilość bloków | Ilość uczonych elementów | Ilość neuronów w danej warstwie |
| Ilość wątków | Ilość neuronów w danej warstwie | Ilość neuronów w poprzedniej warstwie |
| Użycie shared memory | Błąd wyjścia oraz wagi neuronów kolejnej warstwy | Błąd wyjścia oraz pochodne wyjścia neuronów danej warstwy |

W tabeli zostało podane, w jakich sytuacjach jest używany dany kernel, od czego zależy ilość bloków oraz wątków w bloku, oraz opisane, czy jest użyta pamięć dzielona (a jeśli tak, to co  jest w niej zapisywane). Jak widać, w kernelu updateWeightsInTrainingKernel ilość bloków i wątków jest określana w inny sposób niż w innych kernelach. Zostało to tak zrobione, żeby przyspieszyć dostęp do pamięci globalnej.

Kernele są wykonywane przez wiele wątków na raz, a często pamięć wykorzystywana przez te wątki jest przez nie współdzielona. Jeśli dana komórka pamięci może być w jednym momencie czasu zapisywana przez jeden wątek i jednocześnie odczytywana przez inny, program staje się niedeterministyczny – uruchomienie programu wiele razy może dawać różne wyniki (może wystąpić zależność „read after write”, szerzej opisana w (Pfeiffer, 2005)). W kernelach użytych w CNT jest wiele tego typu zależności przy czytaniu/zapisywaniu do pamięci dzielonej. Aby mieć pewność, że zależne od siebie operacje zawsze będą wykonane w odpowiedniej kolejności, użyta jest komenda synchronizująca \_\_syncthreads*[[31]](#footnote-31)*.

### Użyte optymalizacje kerneli

Pierwsza wersja pliku TrainNetwork.cu zawierała niezoptymalizowane kernele służące do wykonywania i uczenia sieci. Używając programu CudaProf autor zauważył, że przy krótko działających kernelach, czas poświęcony na uruchomienie kernela jest często większy niż jego wykonania. Jest to problemem i powoduje, że wywołania niewielkich kerneli na GPU mogą działać dużo dłużej niż ich odpowiedniki na CPU.

W czasie dokonywaniu modyfikacji (optymalizacji) kerneli wykonywanych na GPU, dobrze jest je przetestować na kilku różnych zestawach danych wejściowych – często się zdarza, że modyfikacja w jednym przypadku przyspiesza działanie kernela, a w przypadku innych danych wejściowych je spowalnia. W przypadku niniejszej biblioteki, w czasie optymalizacji, nacisk był kładziony na jak najlepsze zoptymalizowanie działania kerneli w przypadku danych wejściowych o dużych rozmiarach[[32]](#footnote-32). Każdy z poniższych akapitów opisana jest jedna z użytych optymalizacji użytych, aby przyśpieszyć działanie kerneli przy dużej ilości danych.

W rozdziale 2.1.2 jest opisane, czym są tzw. złączone (coalesced) zapisy/odczyty pamięci globalnej, które są wymagane do uzyskania wysokiej przepustowości komunikacji między pamięcią graficzną a GPU. Aby zwiększyć ilość złączonych dostępów do pamięci w bibliotece CNL, tablice zawierające dane wejść testów, wyjścia testów oraz wyjścia każdej warstwy – są wyrównane (aligned) do 16 elementów. Oznacza to, że dane reprezentujące każdy test lub neuron zawsze zaczynają się od elementu o indeksie będącym wielokrotnością 16 elementów (niektóre elementy mogą nie być używane). Wagi sieci nie są wyrównane do 16 elementów, ale odczyty tych wag są złączone (wagi te najpierw są przepisywane do pamięci dzielonej, a później odczytywane).

W kernelach calculateErrorInNotLastLayerKernel i executeLayerKernel 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 dwa razy większej ilości obliczeń. Jednak zmiana prędkości działania kernela executeLayerKernel po tej optymalizacji zależy bardzo od wielkości sieci. W przypadku sieci, gdzie 2 kolejne warstwy mają po 500 neuronów, przyspieszenie jest prawie dwukrotne, natomiast w przypadku sieci, gdzie 2 kolejne warstwy mają po 20 neuronów, kernel działa ok. 10% wolniej. Optymalizacja została użyta w końcowej wersji biblioteki, ponieważ przyspiesza jej działanie w przypadku dużych sieci neuronowych.

Pamięć dzielona (shared memory) jest o wiele szybsza w dostępie od pamięci globalnej (nawet przy złączonym dostępie) i ma dużo krótszy czas dostępu (2 cykle zegara w porównaniu z kilkuset cyklami). Oprócz tego umożliwia łatwe dzielenie się danymi i komunikację między różnymi wątkami wewnątrz bloku. Zmniejszenie ilości operacji na pamięci globalnej i operowanie na szybszej pamięci jest opisywane w poradnikach o technologii CUDA jako pierwszy krok optymalizacji kerneli. Kernele executeLayerKernel, calculateErrorInNotLastLayerKernel, updateWeightsInTrainingKernel używają pamięci dzielonej do przechowywania wag, wejść neuronów, błędów na wyjściu neuronu oraz pochodnej wyjścia neuronu. Jako przykład, poniżej umieściłem fragment kodu kernela executeLayerKernel. Jedną z pierwszych operacji jest kopiowanie globalnych danych do tablicy w pamięci dzielonej. Następnie konieczne jest zsynchronizowanie między wszystkimi wątkami jednego bloku[[33]](#footnote-33). W dwóch pozostałych kernelach odbywa się to w bardzo podobny sposób.

|  |
| --- |
| // 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(); |

Kernel executeLayerKernel używa pamięci dzielonej – jest ona przeznaczana do przechowywania wejść oraz części 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 wszystkich wag nie przekroczy maksymalnej pamięci dzielonej jednego multiprocesora (16 KB). 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 oraz mniej synchronizacji, ale jednocześnie może być mniejsza ilość bloków równocześnie wykonywanych przez multiprocesor GPU. W funkcji executeLayerCUDA jest obliczana optymalna ilość pamięci dzielonej dla wag. Obliczenia te bazują na testach empirycznych – zostały dopasowane tak, żeby kernel działał szybko w różnych konfiguracjach danych wejściowych.

W przypadku uczenia sieci, każde wywołanie executeLayerKernel dla pierwszej warstwy sieci wymaga przekazania tablicy indeksów testów, które mają być użyte w danym kroku uczenia. Zwykle wielkość tej tablicy nie jest duża, ale nie może ona być przekazana jako parametr wywołania kernela – biblioteka CUDA nie pozwala na to. Jednym z rozwiązań byłoby wczytywanie tych indeksów 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 – pamięci stałej (constant). Dzięki temu, że ilość pamięci użytej na przechowanie tabeli indeksów jest mniejsza niż ilość cache wewnątrz każdego multiprocesora (8 KB), dane te będą kopiowane tylko raz do multiprocesora, a później kopiowane z szybkiej pamięci cache.

### Ograniczenia wykonanego algorytmu

Chociaż GPU pozwalają na wykonywanie programów niegraficznych, to cały czas ich głównym zastosowaniem jest rendering grafiki w czasie rzeczywistym. Architektura GPU i różnego typu pamięci graficznych jest głównie przygotowana do tych zastosowań – przez co programy ogólnego zastosowania wykonywane na GPU muszą brać pod uwagę pewne ograniczenia. Niektóre ograniczenia biblioteki CNL są właśnie związane z ograniczeniami technologii CUDA. Te ograniczenia to:

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

Możliwe byłoby ominięcie tych ograniczeń, ale zmiana kodu byłaby żmudna, a modyfikacje mogłyby zwolnić działanie programu.

Pamięć GPU jest zwykle mniejsza niż RAM, a wszystkie dane muszą się na niej zmieścić, jednak nie jest to problemem w przypadku niniejszej biblioteki. Maksymalna możliwa wielkość jednocześnie zadeklarowanej pamięci graficznej wynosi około 140 MB, czyli mniej od minimalnej wymaganej pamięci graficznej potrzebnej do używania CUDA (256 MB) – w przypadku 65535 testów, przy dwóch kolejnych warstwach zawierających 511 neuronów, ilość zadeklarowanej pamięci graficznej wynosi nieco powyżej 65535 \* 511 \* 4 B = 128 MB.

## Testy implementacji sieci MLP

Jednym z zadań w ramach pisania tej pracy było wykorzystanie zestawów testów do uczeniu sieci neuronowej. Dzięki temu pozwoliło to autorowi:

* określić, czy uczenie przez CPU i GPU daje takie same wyniki,
* określić najlepsze parametry uczenia sieci,
* porównać prędkość działania operacji na CPU i GPU oraz prędkość działania na różnych platformach.

Dla zwiększenia wiarygodności wyników, każdy scenariusz zostaje przeprowadzony 3 razy a statystyki błędów oraz czasy działania operacji są uśredniane.

### Opis danych testowych

Zostały wybrane następujące zestawy testów:

* „Concrete” – baza z 7 wejściami opisującymi skład oraz wiek betonu, oraz parametrem wyjściowym określającym jego twardość;
* “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 piersi jest łagodny czy złośliwy.

Wszystkie te trzy zestawy posiadają tylko jedną zmienną wyjściową i nie mają brakujących wartości. Zostały one ściągnięte z ogólnodostępnego repozytorium (Asuncion & Newman, 2010).

### Środowisko testowe

Wszystkie testy zostały przeprowadzone na laptopie z następującą konfiguracją:

* procesor Intel Core 2 Duo P8400, 2.26Ghz,
* 4 GB RAM DDR2, 800Mhz,
* karta graficzna z chipsetem NVIDIA GeForce 9600M GT, 512 MB RAM DDR2,
* Windows Vista 32bit SP2 oraz Windows 7 64bit[[34]](#footnote-34),
* CUDA Toolkit 2.3, CUDA SDK 2.3[[35]](#footnote-35).

### Wpływ parametrów sieci na jakość uczenia

W poniższej tabeli zostały zamieszczone wyniki jakości uczenia sieci neuronowych dla trzech powyższych zestawów testów. Wyniki te zostały wygenerowane przez funkcję makeTrainingWithManyPossibilities, która trenuje sieci neuronowe przy użyciu różnych kombinacji parametrów uczenia. Sieć MLP używana w tej funkcji zawiera jedną warstwę ukrytą z sigmoidalną funkcją aktywacji.

Wyjścia sieci dla wszystkich tych konfiguracji przy przeprowadzaniu uczenia sieci na GPU były praktycznie identyczne do tych przeprowadzanych na hoście (różnice wynosiły najwyżej 0.000001).

W pierwszym wierszu zostały podane informacje o statystykach błędów przy uruchomieniu sieci przed uczeniem (dane oznaczone kursywą). Są tu też podane statystyki błędów sieci z wszystkimi 24 kombinacjami różnych wartości parametrów:

* ilości iteracji w czasie uczenia sieci (40000, 80000 lub 160000),
* współczynnik uczenia eta (0.01 lub 0.03),
* ilość testów użytych w czasie jednej iteracji (1 lub 4),
* ilość neuronów w warstwie ukrytej sieci (32 lub 64).

W kolumnach został umieszony maksymalny i średni bezwzględny błąd przy uczeniu danych z pliku Concrete\_Data.csv, oraz część niepoprawnie sklasyfikowanych testów dla danych z plików iris.data i wdbc.data. Kolorem żółtym oznaczono najlepsze kombinacje parametrów (o najmniejszych błędach) dla każdej z trzech ilości iteracji, w których był użyty jeden test w iteracji (koloru zielonego użyto odpowiednio przy kombinacjach z 4 testami w iteracji). Zestawy danych iris.data i wdbc.data są zadaniami klasyfikacji, w związku z czym ilość błędów (niepoprawnie zaklasyfikowanych elementów) dla różnych konfiguracji sieci często się powtarza.

|  |
| --- |
| Tabela . Jakość uczenia sieci w zależności od różnych parametrów |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Ilość iteracji | Eta | Testów w iteracji | Neurony w warstwie ukrytej | Concrete\_Data.csv | | iris.data | wdbc.data |
| Maks. błąd | Średni błąd |
| *1* | *0.01* | *1* | *32* | *1.055382* | *0.355913* | *0.666667* | *0.542472* |
| 40000 | 0.01 | 1 | 32 | 0.881323 | 0.210257 | 0.022222 | 0.025190 |
| 40000 | 0.01 | 1 | 64 | 0.948090 | 0.226863 | 0.022222 | 0.025776 |
| 40000 | 0.01 | 4 | 32 | 0.862221 | 0.208165 | 0.022222 | 0.016989 |
| 40000 | 0.01 | 4 | 64 | 0.865360 | 0.211914 | 0.020000 | 0.017575 |
| 40000 | 0.03 | 1 | 32 | 0.885246 | 0.213265 | 0.017778 | 0.019332 |
| 40000 | 0.03 | 1 | 64 | 0.971700 | 0.235784 | 0.022222 | 0.019332 |
| 40000 | 0.03 | 4 | 32 | 0.749446 | 0.144566 | 0.020000 | 0.016403 |
| 40000 | 0.03 | 4 | 64 | 0.844297 | 0.147203 | 0.020000 | 0.016403 |
| 80000 | 0.01 | 1 | 32 | 0.874994 | 0.209155 | 0.026667 | 0.019918 |
| 80000 | 0.01 | 1 | 64 | 0.906669 | 0.214904 | 0.026667 | 0.024605 |
| 80000 | 0.01 | 4 | 32 | 0.817314 | 0.186735 | 0.020000 | 0.013474 |
| 80000 | 0.01 | 4 | 64 | 0.913231 | 0.216725 | 0.020000 | 0.014060 |
| 80000 | 0.03 | 1 | 32 | 0.883803 | 0.201784 | 0.020000 | 0.018746 |
| 80000 | 0.03 | 1 | 64 | 0.924605 | 0.211393 | 0.024444 | 0.017575 |
| 80000 | 0.03 | 4 | 32 | 0.718555 | 0.117593 | 0.017778 | 0.011716 |
| 80000 | 0.03 | 4 | 64 | 0.688783 | 0.117480 | 0.020000 | 0.011131 |
| 160000 | 0.01 | 1 | 32 | 0.867153 | 0.208844 | 0.024444 | 0.016989 |
| 160000 | 0.01 | 1 | 64 | 0.876605 | 0.212444 | 0.020000 | 0.017575 |
| 160000 | 0.01 | 4 | 32 | 0.771856 | 0.140492 | 0.020000 | 0.012888 |
| 160000 | 0.01 | 4 | 64 | 0.785761 | 0.146911 | 0.020000 | 0.012302 |
| 160000 | 0.03 | 1 | 32 | 0.761427 | 0.144541 | 0.022222 | 0.015817 |
| 160000 | 0.03 | 1 | 64 | 0.859998 | 0.156688 | 0.020000 | 0.016403 |
| 160000 | 0.03 | 4 | 32 | 0.677923 | 0.123546 | 0.017778 | 0.009373 |
| 160000 | 0.03 | 4 | 64 | 0.659576 | 0.113889 | 0.020000 | 0.008787 |

Z powyższej tabeli można wyciągnąć następujące wnioski na temat wpływu czynników na jakość uczenia sieci:

* W wynikach testów można zauważyć prawidłowość, że najlepsze kombinacje parametrów z 4 testami w iteracji dają lepsze wyniki (mniejszy błąd) niż te z 1 testem w iteracji;
* Wszystkie sieci neuronowe po nauczeniu wykazywały mniejszy błąd niż sieć nienauczona, a większa ilość iteracji uczenia zmniejsza błąd wyjścia sieci;
* W przypadku podanych wyżej testów, współczynnik uczenia eta o wartości 0.03 dawał lepsze wyniki niż współczynnik 0.01;
* Trudno określić wpływ ilości neuronów w warstwie ukrytej; Czasem większa ilość zwiększa, czasem zmniejsza błąd.

### Wpływ optymalizacji na wydajność kerneli

Głównym celem pisania programów na karty graficzne jest uzyskanie większej prędkości niż przy uruchamianiu ich na CPU. Aby rezultaty były zadowalające, często trzeba dokonać odpowiednich optymalizacji programu. Poniżej zamieszczono porównanie prędkości działania uruchamiania sieci neuronowej na zestawach testów dla niezoptymalizowanej oraz zoptymalizowanej wersji kernela GPU, oraz dla wersji CPU. Testy zawierają dwa wejścia oraz dwa wyjścia, a sieć MLP ma dwie warstwy ukryte o tej samej ilości neuronów sigmoidalnych (liczba w pierwszej kolumnie poniższej tabeli określa ilość tych neuronów).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Tabela . Porównanie czasu uruchomienia sieci na CPU i GPU | | | | | |
| Ilość neuronów | Czas uruchamiania sieci na CPU | Całkowity czas uruchamiania sieci na GPU (executeNetworkGPU) | | Czas wywołania samego kernela | |
| Niezoptymalizowane | Zoptymalizowane | Niezoptymalizowany (executeLayerKernel\_OLD) | Zoptymalizowany (executeLayerKernel) |
| 32 | 143 | 239 | 271 | 19 | 5 |
| 64 | 241 | 312 | 241 | 57 | 11 |
| 128 | 494 | 486 | 302 | 237 | 78 |
| 256 | 1274 | 1320 | 722 | 1074 | 449 |
| 510 | 4462 | 3701 | 2000 | 3457 | 1771 |
| 511 | 4492 | 6396 | 2090 | 6169 | 1865 |

Można zauważyć, że czas alokacji i dealokacji pamięci (różnica między czasem wykonania metody executeNetworkGPU a sumą czasów wykonania samych kerneli) nie jest zależna od ilości zadeklarowanych danych i wynosi ok. 200 ms. Czas działania zoptymalizowanego algorytmu na GPU przy małej ilości neuronów jest podobny do tego wykonywanego na hoście, ale im więcej neuronów, tym różnica się zwiększa na korzyść algorytmu GPU. Przy maksymalnej wielkości sieci, wersja CUDA jest 2.15 razy szybsza od wersji CPU (oraz ok. 1.8 razy szybsza od wersji niezoptymalizowanej). Co ciekawe, wersja niezoptymalizowana jest bardzo wrażliwa na ilość danych wejściowych – czas działania dla 511 neuronów jest prawie 2 razy większy niż dla 510. Prawdopodobnie jest to spowodowane tym, że przy 510 neuronach, dostęp do pamięci globalnej był częściej złączony (coalesced).

### Porównanie wydajności uczenia wersji CPU i GPU

Na zestawie danych Concrete\_Data.csv zostało przeprowadzone porównanie prędkości uczenia sieci. Tak jak w rozdziale 4.4.4, sieć ta zawiera 2 warstwy ukryte o tej samej ilości neuronów, a ilość wejść i wyjść testów jest równa dwa. W następującej tabeli zostały umieszczone obliczone czasy w milisekundach 100 iteracji uczenia tej sieci, z podziałem na czas uruchomienia na procesorze CPU oraz GPU, oraz na platformę (Win32 oraz Win64):

|  |
| --- |
| Tabela . Porównanie czasu uczenia sieci na platformach 32 i 64-bitowych |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Ilość neuronów* | *Testów w iteracji* | *Czas uczenia sieci na CPU w ms.* | | *Czas uczenia sieci na GPU w ms.* | |
| *x86* | *x64* | *x86* | *x64* |
| 32 | 1 | 7 | 10 | 142 | 233 |
| 64 | 1 | 19 | 10 | 123 | 114 |
| 128 | 1 | 63 | 46 | 187 | 234 |
| 256 | 1 | 250 | 182 | 389 | 571 |
| 510 | 1 | 1086 | 769 | 1158 | 452 |
| 32 | 16 | 98 | 88 | 116 | 99 |
| 64 | 16 | 289 | 223 | 157 | 145 |
| 128 | 16 | 997 | 774 | 238 | 275 |
| 256 | 16 | 4099 | 2984 | 650 | 1035 |
| 510 | 16 | 19835 | 12470 | 2498 | 764[[36]](#footnote-36) |

Łatwo jest zauważyć zależność, że czas uczenia na CPU jest proporcjonalny do kwadratu ilości neuronów i do ilości testów w iteracji, natomiast przy uczeniu na GPU dużo trudniej znaleźć jakąś zależność. Zwiększenie ilości testów w iteracji nie zwiększa w dużym stopniu czasu działania algorytmu – jest to związane z tym, że różne testy w iteracji mogą być jednocześnie przetwarzane na różnych mikroprocesorach GPU. Dzięki temu w przypadku dużej ilości neuronów w obu warstwach ukrytych i dużej ilości testów w jednej iteracji, uczenie na GPU jest ok. 8 razy szybsze niż wykonane na CPU. Jest to większe przyspieszenie niż w przypadku uruchomienia sieci.

Na platformie 64-bitowej można zauważyć przyspieszenie 20-30% w wykonywaniu operacji na CPU oraz spowolnienie w przypadku wykonania na GPU. Przyspieszenie na CPU może być tłumaczone tym, że program jest wykonywany na procesorze z architekturą 64-bitową, dzięki czemu operacje na 64-bitowych liczbach zmiennoprzecinkowych są wykonywane szybciej. Spowolnienie w wykonaniu programu na GPU może być spowodowane nieoptymalnie napisanym sterownikiem graficznym na tę platformę.

### Informacje końcowe

W związku z tym, że testy zostały przeprowadzone na laptopie, procesor CPU oraz karta graficzna są dużo mniej wydajne od najszybszych obecnie dostępnych układów – w związku z tym na komputerze z wysoko wydajnym procesorem oraz GPU, wyniki powyższych testów byłyby zdecydowanie inne niż te zaprezentowane w powyższych tabelach. Należy również wziąć pod uwagę, że maksymalną wydajność na najszybszych obecnie GPU (takich jak NVIDIA GeForce GTX 285) można uzyskać dopiero przy około 100 testach na iterację użytych w czasie uczenia sieci.

W programie nie ma możliwości zrównoleglenia obliczeń na CPU. Oprócz tego, przy porównywaniu prędkości CPU i GPU trzeba wziąć pod uwagę, że obliczenia na CPU są robione przy pomocy klasy std::vector, która jest wolniejsza od zwykłych tablic.

Różnice między końcowymi rezultatami uczenia sieci pomiędzy CPU a GPU zwykle są bardzo małe – oznacza to, że użycie zmiennych pojedynczej precyzji do wykonywania operacji na GPU nie pogorszyło to w dużym stopniu dokładności obliczeń. Jednak przy dużej wartości współczynnika uczenia eta, dużej wielkości sieci neuronowej oraz dużej ilości iteracji, te różnice między rezultatami zaczynają się zwiększać.

## Zewnętrzne biblioteki

W implementacji biblioteki CNL użyłem dwóch bibliotek zewnętrznych – Mersenne Twister i TinyXML, opisanych poniżej.

Generator liczb pseudolosowych wbudowany w język C++, funkcja rand, jest bardzo niskiej jakości – działa wolno, a wylosowane liczby nie spełniają wymagań wielu testów na losowość (przez to badania wykonane z tym generatorem mogą nie dawać wiarygodnych rezultatów). Dla zadań wykonywanych w tej bibliotece, bardzo dobrym rozwiązaniem jest generator Mersenne Twister, który jest polecany do programów ogólnego użytku. Wybrałem bibliotekę obiektową stworzoną przez Richarda J. Wagnera, opisaną w (Wagner, 2009).

Jako format przechowywania danych sieci i zestawów testów, wybrałem XML. Zależało mi na znalezieniu biblioteki do obsługi XML, która przedstawia dokument w postaci struktury obiektów[[37]](#footnote-37), oraz jest prosta w obsłudze. TinyXML to jedna z najpopularniejszych darmowych bibliotek dla C++, opisana w (Thomason).

## Narzędzia pomocnicze

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

* czas wykonania przez CPU, GPU,
* ilość danych zapisanych lub odczytanych z pamięci globalnej, z podziałem na operacje złączone (coalesced) i nie złączone,
* ilość danych zapisanych lub odczytanych z pamięci dzielonej,
* ilość wykonanych instrukcji.

Poniżej jest zrzut ekranu programu:

|  |
| --- |
|  |
| Rysunek 21. Program CUDA Visual Profiler |

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

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

## Możliwości rozwoju programu

Biblioteka CNL obsługuje wszystkie najważniejsze operacje na sieci MLP oraz zestawie testów, jednak mogłaby być poprawiona na wiele sposobów:

* Dla lepszej oceny wiarygodności jakości uczenia, można by dzielić zestaw testów na zestaw uczący oraz testujący (weryfikujący);
* Dodanie możliwości zmiany współczynnika uczenia w trakcie procesu uczenia (często daje to lepsze wyniki uczenia niż jedna stała wartość);
* Dodanie obsługi sieci neuronowe RBF, Hopfielda, lub innej do istniejącego kodu;
* Zmiana w kernelach z operatora mnożenia liczb całkowitych na użycie funkcji \_\_[u]mul24, która zwiększyłaby w niewielkim stopniu prędkość ich działania[[38]](#footnote-38);
* Aktualnie istnieje ograniczenie na ilość neuronów w warstwie do 511 oraz na ilość testów w zestawie testów do 65535 (opisane w rozdziale 4.3.4). Można by je usunąć, ale spowodowałoby to spadek wydajności;
* Dodanie obsługi kilku równolegle działających GPU;
* Dodanie obsługi wykonania części CPU na kilku rdzeniach lub procesorach.

# Wnioski

Jak wspomniano w niniejszej pracy, na procesorach GPU można wydajnie wykonywać zadania dające się łatwo zrównoleglić. Uruchamianie oraz uczenie sieci neuronowych są takimi właśnie procesami, a w tej pracy zostało pokazane, że prędkość działania tego typu aplikacji uruchomionej na GPU może być większa niż na CPU. Przyspieszenie działania w stosunku do wersji uruchamianej na CPU zostało osiągnięte dzięki kolejnym optymalizacjom i usuwaniu wąskich gardeł algorytmu. Należy pamiętać jednak o tym, że aby osiągnąć przyspieszenie działania na GPU, trzeba użyć problemów o dużych rozmiarach (zostało to zauważone również w pracach innych autorów na temat GPGPU).

W czasie opracowywania biblioteki CNL autor zauważył, że  napisanie programu w technologii CUDA nie jest trudne, natomiast dużo większy kłopot sprawia zaprojektowanie go tak, by osiągał on maksymalną wydajność. Trudności spowodowane są tym, że program może być napisany na wiele sposobów oraz może używać wielu różnych typów pamięci, a dla osiągnięcia wysokiej prędkości transferu, dostępy do pamięci muszą być wykonywane w odpowiedni sposób. W przypadku opisywanej biblioteki, dodatkowym utrudnieniem przy projektowaniu algorytmów był fakt, że rozmiar danych przetwarzanych na GPU nie jest znany w czasie kompilacji.

~~Gdybym teraz miał za zadanie zrobić ten program jeszcze raz, starałbym się uprościć kernele i zmniejszyć ilość instrukcji skoku – gdyż zwalniają one działanie programu w dużo większym stopniu niż się spodziewałem. 🡪 Można zauważyć, że uproszczenie kerneli i zmniejszenie ilości instrukcji skoku przyspieszają one działanie programu w znacznym stopniu.~~

~~(?????) Widać, że w przypadku sieci neuronowych użycie float nie pogarsza jakości wyników programu.~~

Autor uważa, że użycie CUDA do wykonania opisywanego programu było dobrym wyborem. Nauczenie się tej technologii nie było trudne, wszystkie pojawiające się problemy można było rozwiązać czytając dokumentację lub zasięgając pomocy na forum NVIDIA, a w optymalizacji biblioteki CNL pomocne było specjalne narzędzie profilujące kod GPU.

Firma NVIDIA w swej pracy przykłada dużą wagę do technologii CUDA. Nowa architektura kart graficznych Fermi dostarcza wielu zmian w zakresie łatwości programowania. Jak wykazują ostatnie lata, moc obliczeniowa GPU zwiększa się wyraźnie szybciej niż w przypadku procesorów CPU. Jeśli ten trend w przyszłości się utrzyma, zastosowanie GPGPU będzie dawało coraz więcej korzyści i przyczyni się to do zwiększenia jego popularności. Niestety wysoka moc procesorów graficznych jest okupiona większym zużyciem prądu. Inną sprawą, która może zniechęcić programistów jest konieczność pisania większej liczby linii kodu spowodowanej wymaganymi kopiowaniami danych między RAM a pamięcią graficzną. Jednak autor uważa, że ze względu na wcześniej wymienione korzyści, technologia GPGPU może być bardzo przydatna w wielu sytuacjach (i w dziedzinie superkomputerów, i w programach uruchamianych na zwykłych komputerach).

# Bibliografia

AMD. (brak daty). *ATI Stream Technology.* Pobrano z lokalizacji AMD: http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx

Asuncion, A., & Newman, D. (2010). *UCI Machine Learning Repository.* Pobrano z lokalizacji Irvine, CA: University of California, School of Information and Computer Science: http://mlr.cs.umass.edu/ml

Baggio, D. L. (2007). *GPGPU Based Image Segmentation Livewire Algorithm Implementation.* Pobrano z lokalizacji GPGPU.org: http://gpgpu.org/2008/04/01/gpgpu-based-image-segmentation-livewire-algorithm-implementation

Berillo, A. (2008, Październik 21). *NVIDIA CUDA - Non-graphic computing with graphics processors.* Pobrano z lokalizacji iXBT Labs: http://ixbtlabs.com/articles3/video/cuda-1-p1.html

Breitbart, J. (2008, Sierpień 7). *Case studies on GPU usage and data structure design.* Pobrano z lokalizacji Geeks3D: http://www.geeks3d.com/downloads/200808/Jens\_Breitbart\_thesis.pdf

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., i inni. (2004). *Brook for GPUs: Stream Computing on Graphics Hardware.* Pobrano z lokalizacji CiteSeerX: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1.5420&rep=rep1&type=pdf

Cristianini, N. (2007). *Support Vector Machines - Online References for Further Reading.* Pobrano z lokalizacji Support Vector Machines Homepage: http://www.support-vector.net/references.html

Fung, J., & Murray, T. (2008, Grudzień). *Building CUDA Photoshop Filters for the GPU.* Pobrano z lokalizacji NVIDIA: http://developer.download.nvidia.com/compute/cuda/Photoshop/CUDAFilters4.pdf

GPGPU.org. (brak daty). *About GPGPU.org.* Pobrano z lokalizacji GPGPU.org: http://gpgpu.org/about

Harris, M., & Buck, I. (2005, Kwiecień). *GPU Gems 2, Chapter 34. GPU Flow-Control Idioms.* Pobrano z lokalizacji NVIDIA Developer Zone: http://http.developer.nvidia.com/GPUGems2/gpugems2\_chapter34.html

Harvey, J. P. (2009, Wrzesień). *GPU acceleration of object classification algorithms using NVIDIA CUDA.* Pobrano z lokalizacji Digital media library - RIT Scholars: https://ritdml.rit.edu/handle/1850/10894

Jayram, M. N., Dutt, N., Krichmar, J. L., Nicolau, A., & Veidenbaum, A. (2009). *Efficient Simulation of Large-Scale Spiking Neural Networks Using CUDA Graphics Processors.* Pobrano z lokalizacji Jayram, Moorkanikara Nageswaran - Home Page: http://www.ics.uci.edu/~jmoorkan/pub/gpusnn-ijcnn.pdf

Jorden. (2010, Kwiecień 17). *FAQ: Nvidia CUDA & ATI Stream (CAL) FAQ.* Pobrano z lokalizacji BOINC: http://boincfaq.mundayweb.com/index.php?language=1&view=471

Katedra Inżynierii Komputerowej P.Cz. (2004). *Zastosowanie sieci neuronowych.* Pobrano z lokalizacji Katedra Inżynierii Komputerowej Politechniki Częstochowskiej: http://www.kik.pcz.czest.pl/nn/zastosowanie.php

Klaus, R. (brak daty). *Budowa Neuronu.* Pobrano z lokalizacji Rafał Klaus - Strona internetowa: http://www.cs.put.poznan.pl/rklaus/assn/neuron.htm

Kołton, M., & Kwiatkowski, M. (2005, Czerwiec). *Sieć Hopfielda.* Pobrano z lokalizacji Wirtualne Laboratorium Sztucznej Inteligencji: http://galaxy.agh.edu.pl/~vlsi/AI/hopf/hopfield\_pl.html

Lavignotte, F. (2010, Marzec 23). *CUDA Toolkit 3.0 released.* Pobrano z lokalizacji NVIDIA Forums: http://forums.nvidia.com/index.php?showtopic=163511&st=20&p=1025290&#entry1025290

Longbottom, R. (2009, Listopad). *Dhrystone Benchmark Results On PCs.* Pobrano z lokalizacji Roy Longbottom's PC Benchmark Collection: http://www.roylongbottom.org.uk/dhrystone%20results.htm

Mattb3. (2010, Kwiecień 14). *Poor cgemm performance with cuda 3.0.* Pobrano z lokalizacji NVIDIA Forums: http://forums.nvidia.com/index.php?showtopic=166184&st=0&p=1039424&#entry1039424

Nasse, F., Thurau, C., & Fink, G. A. (2009). *Face Detection Using GPU-Based Convolutional Neural Networks .* Pobrano z lokalizacji Google Books: http://books.google.com/books?id=g-sIocvOrUUC&pg=PA83&lpg=PA83&dq=%22+Face+Detection+Using+GPU-Based+Convolutional+Neural+Networks+%22&source=bl&ots=HQhMY3wI10&sig=tOmcxv7eb5\_2uSBHzsRaO9IpkVg&hl=pl&ei=4nnpS5bHHMigOPm\_lf8K&sa=X&oi=book\_result&ct=result&res

NVIDIA. (2010, Kwiecień 21). *CUDA 3.0 Downloads.* Pobrano z lokalizacji NVIDIA: http://developer.nvidia.com/object/cuda\_3\_0\_downloads.html

NVIDIA. (brak daty). *CUDA GPU Computing.* Pobrano z lokalizacji NVIDIA Forums: http://forums.nvidia.com/index.php?showforum=62

NVIDIA. (brak daty). *CUDA-Accelerated Applications.* Pobrano z lokalizacji NVIDIA: http://www.nvidia.com/object/cuda\_app\_tesla.html

NVIDIA. (brak daty). *GeForce 256.* Pobrano z lokalizacji NVIDIA: http://www.nvidia.com/page/geforce256.html

NVIDIA. (2010, Luty 8). *MATLAB plug-in for CUDA.* Pobrano z lokalizacji NVIDIA Developer Web Site: http://developer.nvidia.com/object/matlab\_cuda.html

NVIDIA. (2009, Sierpień 26). *NVIDIA CUDA Programming Guide 2.3.* Pobrano z lokalizacji NVIDIA: http://developer.download.nvidia.com/compute/cuda/2\_3/toolkit/docs/NVIDIA\_CUDA\_Programming\_Guide\_2.3.pdf

NVIDIA. (2010, Luty 20). *NVIDIA CUDA Programming Guide 3.0.* Pobrano z lokalizacji NVIDIA: http://developer.download.nvidia.com/compute/cuda/3\_0/toolkit/docs/NVIDIA\_CUDA\_ProgrammingGuide.pdf

NVIDIA. (brak daty). *NVIDIA PhysX.* Pobrano z lokalizacji NVIDIA Polska: http://www.nvidia.pl/object/nvidia\_physx\_pl.html

NVIDIA. (2009). *NVIDIA’s Next Generation CUDA Compute Architecture: Fermi, v1.1.* Pobrano z lokalizacji NVIDIA: http://www.nvidia.com/content/PDF/fermi\_white\_papers/NVIDIA\_Fermi\_Compute\_Architecture\_Whitepaper.pdf

NVIDIA. (brak daty). *Supercomputing at 1/10th the Cost.* Pobrano z lokalizacji NVIDIA: http://www.nvidia.com/object/tesla\_computing\_solutions.html

NVIDIA. (brak daty). *What is CUDA?* Pobrano z lokalizacji NVIDIA: http://www.nvidia.com/object/what\_is\_cuda\_new.html

Osowski, S. (1996). *Sieci neuronowe w ujęciu algorytmicznym.* Warszawa: Wydawnictwa Naukowo-Techniczne.

Pfeiffer, J. (2005, Luty 23). *CS473 - Pipeline Hazards.* Pobrano z lokalizacji Joe Pfeiffer - Home Page: http://www.cs.nmsu.edu/~pfeiffer/classes/473/notes/hazards.html

Prabhu, R. D. (2007). *GNeuron: Parallel Neural Networks with GPU.* Pobrano z lokalizacji HiPC - International Conference on High Performance Computing: http://www.hipc.org/hipc2007/posters/GNeuron.pdf

Rutkowski, L. (2005). *Metody i techniki sztucznej inteligencji.* Warszawa: Wydawnictwo Naukowe PWN.

Sandhu, T. (2010, Marzec 26). *NVIDIA's GeForce GTX 480 finally unleashed. Reviewed and rated.* Pobrano z lokalizacji HEXUS.net: http://www.hexus.net/content/item.php?item=24000&page=3

Schabauer, H., Schikuta, E., & Weishaupl, T. (2005). *Solving Very Large Traveling Salesman Problems by SOM Parallelization on Cluster Architectures.* Pobrano z lokalizacji ACM Portal: http://portal.acm.org/citation.cfm?id=1110441

Spek, J. v. (2008, Kwiecień 1). *The CUDA Compiler Driver - NVCC.* Pobrano z lokalizacji Frank Mueller Homepage: http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/2.0/nvcc\_2.0.pdf

The Khronos Group. (brak daty). *OpenCL - The open standard for parallel programming of heterogeneous systems.* Pobrano z lokalizacji The Knronos Group: http://www.khronos.org/opencl/

Thomason, L. (brak daty). *Lee Thomason's Homepage .* Pobrano z lokalizacji TinyXML Main Page: http://sourceforge.net/projects/tinyxml

Triolet, D. (2007, Marzec 21). *Nvidia CUDA: preview.* Pobrano z lokalizacji BeHardware: http://www.behardware.com/articles/659-1/nvidia-cuda-preview.html

W3C. (2009, Styczeń 6). *W3C Document Object Model.* Pobrano z lokalizacji World Wide Web Consortium (W3C) Website: http://www.w3.org/DOM

Wagner, R. J. (2009, Wrzesień 28). *Mersenne Twister Random Number Generator.* Pobrano z lokalizacji Creations of Rick Wagner: http://www-personal.umich.edu/~wagnerr/MersenneTwister.html

Wikipedia. (2010, Maj 8). *Artificial neural network.* Pobrano z lokalizacji Wikipedia: http://en.wikipedia.org/wiki/Artificial\_neural\_network

Wikipedia. (2010, Maj 14). *Comparison of Nvidia graphics processing units.* Pobrano z lokalizacji Wikipedia: http://en.wikipedia.org/wiki/Comparison\_of\_Nvidia\_graphics\_processing\_units

Wikipedia. (brak daty). *GPGPU.* Pobrano z lokalizacji Wikipedia: http://en.wikipedia.org/wiki/GPGPU

Wikipedia. (2010, Maj 6). *Neuron.* Pobrano z lokalizacji Wikipedia: http://en.wikipedia.org/wiki/Neuron

Wikipedia. (2010, Maj 13). *OpenCL.* Pobrano z lokalizacji Wikipedia: http://en.wikipedia.org/wiki/OpenCL

Wikipedia. (2010, Styczeń 1). *Radialna funkcja bazowa.* Pobrano z lokalizacji Wikipedia: http://pl.wikipedia.org/wiki/Radialna\_funkcja\_bazowa

Wikipedia. (brak daty). *Video card.* Pobrano z lokalizacji Wikipedia: http://en.wikipedia.org/wiki/Graphic\_card

Wilk, A. (2009, Październik 5). *Mity i fakty: komputery bez procesora.* Pobrano z lokalizacji Trochetechniki.pl: http://www.trochetechniki.pl/Mity-i-fakty%3A-komputery-bez-procesora,t,929,page,1.html

# Załączniki

Załącznik 1 - Przykłady kodu CUDA

Aktualnie technologia CUDA udostępnia dwa interfejsy programowania: wysokopoziomowy C for CUDA oraz niskopoziomowy CUDA Driver API. Są one wzajemnie wykluczające – program może używać tylko jednego z nich.

Interfejs C for CUDA zawiera pewne rozszerzenia do języka C. Pozwalają one zdefiniować kernel jako funkcję C oraz wyspecyfikować wymiary gridu oraz bloku przy uruchamianiu kernela. Ten interfejs jako jedyny wspiera tryb emulacji. Jest on łatwiejszy w użyciu i popularniejszy wśród programistów.

Interfejs CUDA Driver API jest niskopoziomowym API, nie zawiera on rozszerzeń języka C. Zawiera funkcje do ładowania kerneli jako moduły, ustawiania ich parametrów, oraz do uruchamiania ich .

Poniżej umieściłem przykładowe kody prezentujące sposób programowania obu podanych wyżej interfejsów. Polegają one na uruchomieniu kernela sumującego wartości w dwóch wektorach i zapisującego je do trzeciego wektora. Użyłem fragmentów kodów z CUDA SDK 3.0, usuwając pewne elementy (np. sprawdzanie poprawności wywołania funkcji) dla większej czytelności kodu.

Kod C for CUDA

Poniżej użyłem fragmentu kodu z projektu vectorAdd:

|  |
| --- |
| // Variables  float\* h\_A;  float\* h\_B;  float\* h\_C;  float\* d\_A;  float\* d\_B;  float\* d\_C;  // Device code  \_\_global\_\_ void VecAdd(const float\* A, const float\* B, float\* C, int N)  {  int i = blockDim.x \* blockIdx.x + threadIdx.x;  if (i < N)  C[i] = A[i] + B[i];  }  // Host code  int main(int argc, char\*\* argv)  {  printf("Vector addition\n");  int N = 50000;  size\_t size = N \* sizeof(float);  // Allocate input vectors h\_A and h\_B in host memory  h\_A = (float\*)malloc(size);  h\_B = (float\*)malloc(size);  h\_C = (float\*)malloc(size);    ... Initialize input vectors h\_A and h\_B ...  // Allocate vectors in device memory  cudaMalloc((void\*\*)&d\_A, size);  cudaMalloc((void\*\*)&d\_B, size);  cudaMalloc((void\*\*)&d\_C, size);  // Copy vectors from host memory to device memory  cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice);  cudaMemcpy(d\_B, h\_B, size, cudaMemcpyHostToDevice);  // Invoke kernel  int threadsPerBlock = 256;  int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d\_A, d\_B, d\_C, N);  // Copy result from device memory to host memory  // h\_C contains the result in host memory  cutilSafeCall( cudaMemcpy(h\_C, d\_C, size, cudaMemcpyDeviceToHost) );  ... Verify result ...  } |

Kod CUDA Driver API

Plik vectorAdd.cu z projektu vectorAddDrv, zawierający kernel, ma następującą treść:

|  |
| --- |
| // Device code  extern "C" \_\_global\_\_ void VecAdd(const float\* A, const float\* B, float\* C, int N)  {  int i = blockDim.x \* blockIdx.x + threadIdx.x;  if (i < N)  C[i] = A[i] + B[i];  } |

Przed uruchomieniem programu w trybie CUDA Driver API, wymagana jest kompilacja kerneli do postaci assemblera CUDA (pliki z rozszerzeniem .ptx) lub postaci binarnej (pliki z rozszerzeniem .cubin). Po uruchomieniu, te pliki muszą być załadowane do programu. Poniższy kod jest fragmentem kodu z pliku vectorAdd.cpp:

|  |
| --- |
| // Variables  CUdevice cuDevice;  CUcontext cuContext;  CUmodule cuModule;  CUfunction vecAdd;  float\* h\_A;  float\* h\_B;  float\* h\_C;  CUdeviceptr d\_A;  CUdeviceptr d\_B;  CUdeviceptr d\_C;  // Functions  bool findModulePath(const char\*, string &, char\*\*, string &);  // Host code  int main(int argc, char\*\* argv)  {  printf("Vector Addition (Driver API)\n");  int N = 50000;  unsigned int size = N \* sizeof(float);  CUresult error;  // Initialize  error = cuInit(0);  // Get number of devices supporting CUDA  int deviceCount = 0;  error = cuDeviceGetCount(&deviceCount);  if (deviceCount == 0) { return -1; } // No devices  // Get handle for device 0  error = cuDeviceGet(&cuDevice, 0);  // Create context  error = cuCtxCreate(&cuContext, 0, cuDevice);  // first search for the module path before we load the results  string module\_path, ptx\_source;  if (!findModulePath ("vectorAdd.ptx", module\_path, argv, ptx\_source)) {  if (!findModulePath ("vectorAdd.cubin", module\_path, argv, ptx\_source)) {  printf("> findModulePath could not find <vectorAdd> ptx or cubin\n"); return -1;  }  }  else {  printf("> initCUDA loading module: <%s>\n", module\_path.c\_str());  }  // Create module from binary file (PTX or CUBIN)  if (module\_path.rfind("ptx") != string::npos)  {  // in this branch we use compilation with parameters  const unsigned int jitNumOptions = 3;  CUjit\_option \*jitOptions = new CUjit\_option[jitNumOptions];  void \*\*jitOptVals = new void\*[jitNumOptions];  // set up size of compilation log buffer  jitOptions[0] = CU\_JIT\_INFO\_LOG\_BUFFER\_SIZE\_BYTES;  int jitLogBufferSize = 1024;  jitOptVals[0] = (void \*)jitLogBufferSize;  // set up pointer to the compilation log buffer  jitOptions[1] = CU\_JIT\_INFO\_LOG\_BUFFER;  char \*jitLogBuffer = new char[jitLogBufferSize];  jitOptVals[1] = jitLogBuffer;  // set up pointer to set the Maximum # of registers for a particular kernel  jitOptions[2] = CU\_JIT\_MAX\_REGISTERS;  int jitRegCount = 32;  jitOptVals[2] = (void \*)jitRegCount;  error = cuModuleLoadDataEx(&cuModule, ptx\_source.c\_str(), jitNumOptions, jitOptions, (void \*\*)jitOptVals);  printf("> PTX JIT log:\n%s\n", jitLogBuffer);  } else {  error = cuModuleLoad(&cuModule, module\_path.c\_str());  }  // Get function handle from module  error = cuModuleGetFunction(&vecAdd, cuModule, "VecAdd");  // Allocate input vectors h\_A and h\_B in host memory  h\_A = (float\*)malloc(size);  h\_B = (float\*)malloc(size);  h\_C = (float\*)malloc(size);    ... Initialize input vectors h\_A and h\_B ...  // Allocate vectors in device memory  error = cuMemAlloc(&d\_A, size);  error = cuMemAlloc(&d\_B, size);  error = cuMemAlloc(&d\_C, size);  // Copy vectors from host memory to device memory  error = cuMemcpyHtoD(d\_A, h\_A, size);  error = cuMemcpyHtoD(d\_B, h\_B, size);  // Invoke kernel  #define ALIGN\_UP(offset, alignment) \  (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)  int offset = 0;  void\* ptr;  ptr = (void\*)(size\_t)d\_A;  ALIGN\_UP(offset, \_\_alignof(ptr));  error = cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));  offset += sizeof(ptr);  ptr = (void\*)(size\_t)d\_B;  ALIGN\_UP(offset, \_\_alignof(ptr));  error = cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));  offset += sizeof(ptr);  ptr = (void\*)(size\_t)d\_C;  ALIGN\_UP(offset, \_\_alignof(ptr));  error = cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));  offset += sizeof(ptr);  ALIGN\_UP(offset, \_\_alignof(N));  error = cuParamSeti(vecAdd, offset, N);  offset += sizeof(N);  error = cuParamSetSize(vecAdd, offset);  int threadsPerBlock = 256;  int blocksPerGrid =  (N + threadsPerBlock - 1) / threadsPerBlock;  error = cuFuncSetBlockShape(vecAdd, threadsPerBlock, 1, 1);  error = cuLaunchGrid(vecAdd, blocksPerGrid, 1);  // Copy result from device memory to host memory  // h\_C contains the result in host memory  error = cuMemcpyDtoH(h\_C, d\_C, size);    ... Verify result ...  }  bool inline  findModulePath(const char \*module\_file, string & module\_path, char \*\*argv, string & ptx\_source)  {  module\_path = cutFindFilePath(module\_file, argv[0]);  if (module\_path.empty()) {  printf("> findModulePath could not find file: <%s> \n", module\_file); return false;  } else {  printf("> findModulePath found file at <%s>\n", module\_path.c\_str());  if (module\_path.rfind(".ptx") != string::npos) {  // We load contents of module\_path to the ptx\_source string  FILE \*fp = fopen(module\_path.c\_str(), "rb");  fseek(fp, 0, SEEK\_END);  int file\_size = ftell(fp);  char \*buf = new char[file\_size+1];  fseek(fp, 0, SEEK\_SET);  fread(buf, sizeof(char), file\_size, fp);  fclose(fp);  buf[file\_size] = '\0';  ptx\_source = buf;  delete[] buf;  }  return true;  }  } |

Interfejs CUDA Driver API wymaga od programisty zainicjalizowania biblioteki CUDA oraz programowego ładowania skompilowanego kernela do pamięci, a uruchamianie kernela jest dużo bardziej skomplikowane. To wszystko powoduje, że wymaga więcej kodu do napisania, jest trudniejszy w użyciu, ale pozwala na większy zakres kontroli nad przebiegiem programu.

W konkurencyjnych bibliotekach OpenCL oraz ATI Stream, procedura uruchamiania programu na GPU jest podobna w interfejsie C for CUDA – jest możliwość bezpośredniego wywołania kernela. Różnią się one składnią, ale programy w nich napisane wymagają podobnej liczby linii kodu, co w interfejsie C for CUDA. Przykładowe kody używające tamtych bibliotek są umieszczone w oraz .

Załącznik 2 – Formaty plików

W tym rozdziale są opisane formaty plików używanych przez bibliotekę CNL.

Plik zestawu testów CSV

Plik forestfires2.csv użyty w kodzie z rozdziału 4.1.3 ma następującą zawartość:

|  |
| --- |
| X,Y,month,day,FFMC,area  7,5,mar,fri,86.2,0  7,4,oct,tue,90.0,0  8,4,oct,sat,90.6,1 |

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 między formatami plików tego typu to znak oddzielający elementy – albo przecinek albo średnik.

Pliki CSV użyte w programie nie były stworzone przez autora, więc metoda InputTestSet::loadFromCSVFile zawiera pewne zabezpieczenia przed niepoprawnymi danymi w pliku. Akceptowane są jedynie zestawy testów, w których:

* znajdują się minimalnie 3 testy,
* ilość wartości w każdej linii jest taka sama,
* każda z wartości w każdym teście jest niepusta,
* każda kolumna(atrybut zestawu testów) ma przynajmniej 2 możliwe wartości.

Plik zestawu testów XML

Poniżej jest zamieszczony plik TestSetFromCSV.xml wygenerowany na końcu sekwencji w kodzie z rozdziału 4.1.3:

|  |
| --- |
| <?xml version="1.0" ?>  <TestSet SourceFileName="Resources\Test\_data\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="area" IsOutputAttribute="True" ColumnIndexInInputFile="5" ColumnIndexInStructure="0" IsLiteralAttribute="False" MinValue="0" MaxValue="1" />  </AttributeMappings>  <Tests>  <Test>  <Inputs>7;5;[-1][mar];[1;-1;-1][fri];86.2</Inputs>  <CorrectOutputs>0</CorrectOutputs>  <NetworkOutputs>-2.3013e-005</NetworkOutputs>  <NetworkOutputsGPU>-2.30074e-005</NetworkOutputsGPU>  </Test>  <Test>  <Inputs>7;4;[1][oct];[-1;1;-1][tue];90</Inputs>  <CorrectOutputs>0</CorrectOutputs>  <NetworkOutputs>4.43972e-005</NetworkOutputs>  <NetworkOutputsGPU>4.43459e-005</NetworkOutputsGPU>  </Test>  <Test>  <Inputs>8;4;[1][oct];[-1;-1;1][sat];90.6</Inputs>  <CorrectOutputs>1</CorrectOutputs>  <NetworkOutputs>0.999951</NetworkOutputs>  <NetworkOutputsGPU>0.999951</NetworkOutputsGPU>  </Test>  </Tests>  </TestSet> |

Jak widać, plik w formacie XML jest zdecydowanie większy od pliku CSV. Głównym powodem większego rozmiaru jest to, że każda wartość jest opisana nazwą. Oprócz tego, zestaw testów w formacie XML zawiera dodatkową informację niedostępną w pliku CSV – zawarte są w nim również wyjścia sieci obliczone przez CPU i GPU.

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. Pierwszym potomkiem elementu TestSet jest AttributeMappings – element zawierający 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 – atrybut 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 w zestawie testów 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 true/false określający, czy atrybut jest klasyfikacyjny;
* MinValue – atrybut występuje tylko przy atrybutach liczbowych (IsLiteralAttribute = false); Określa minimalną wartość danej zmiennych w danej kolumnie;
* MaxValue – atrybut występuje tylko przy atrybutach liczbowych (IsLiteralAttribute = false). Określa maksymalną wartość zmiennych w danej kolumnie.

Atrybuty klasyfikacyjny (IsLiteralAttribute = false) mają potomków o nazwie ColumnElementName. Każdy z tych potomków zawiera literalną wartość klasyfikacyjną. Jeśli są możliwe tylko dwie wartości, wtedy danemu atrybutowi testu odpowiada jeden neuron wejściowy/wyjściowy (wartość pierwsza odpowiada wartości ‑1, a druga wartości 1). Jeśli jest więcej wartości, wtedy każda możliwa wartość odpowiada jednemu neuronowi wejściowemu/wyjściowemu. W tym przypadku test, którego ta kolumna ma n-tą wartość, powoduje, że wszystkie neurony wejściowe/wyjściowe 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::trainNetworkGPU lub NeuralNetwork::executeNetworkGPU.

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 jako 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. Np. jeśli dana kolumna wyjściowa przyjmuje wartości ‘a’, ‘b’, ‘c’, a wartości neuronów wyjściowych odpowiadających tej kolumnie wynoszą ‑0.8, 0.3, 0.1, to oznacza, że w tym teście została wybrana druga wartość tej kolumny (czyli ‘b’).

Struktura pliku zestawu testów XML została zaprojektowana tak, żeby był on czytelny dla człowieka. Dlatego niektóre informacje o zestawie testów są nadmiarowe: w elemencie AttributeMapping jest informacja o minimalnej i maksymalnej wartości, a można to też wyczytać z konkretnych wartości testów. Jako potomkowie AttributeMapping jest podana lista elementów klasyfikacyjnych, która znajduje się też w potomku Tests.

Format sieci MLP w XML

Poniżej jest plik sieci MLP stworzony na końcu sekwencji w kodzie z rozdziału 4.1.3 w pliku NetworkStruct.xml:

|  |
| --- |
| <?xml version="1.0" ?>  <NeuralNetwork Type="MLP">  <Layer>  <Neuron NeuronType="NT\_SIGMOID">  <Weights>-0.678326;0.122323;-0.114991;0.121475;0.546221;-0.673357;-0.1967;0.0052644;</Weights>  </Neuron>  <Neuron NeuronType="NT\_SIGMOID">  <Weights>-0.717653;0.112465;-0.123191;0.119085;0.573816;-0.718274;-0.206293;0.0280956;</Weights>  </Neuron>  <Neuron NeuronType="NT\_SIGMOID">  <Weights>-0.500953;0.119529;-0.113412;0.118167;0.418017;-0.500756;-0.167803;-0.0260106;</Weights>  </Neuron>  </Layer>  <Layer>  <Neuron NeuronType="NT\_LINEAR">  <Weights>-1.04635;-1.14307;-0.653026;1.24682;</Weights>  </Neuron>  </Layer>  </NeuralNetwork> |

Główny element 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 warstwę ukrytą lub wyjściową. Warstwy są uporządkowane w kolejności od pierwszej warstwy ukrytej do warstwy wyjściowej. Ilość neuronów wejściowych w testach przyjmowana przez tę sieć 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.

# Spis ilustracji

[Rysunek 1. Model fizyczny CPU i GPU 8](#_Toc263023020)

[Rysunek 2. Hierarchia procesorów oraz typów pamięci w karcie graficznej 9](#_Toc263023021)

[Rysunek 3. Złączone dostępy do pamięci na urządzeniach CC 1.0 i 1.1 11](#_Toc263023022)

[Rysunek 4. Architektura CUDA 14](#_Toc263023023)

[Rysunek 5. Podział pracy na bloki i watki 15](#_Toc263023024)

[Rysunek 6. Proces kompilacji plików źródłowych CUDA 18](#_Toc263023025)

[Rysunek 7. Schemat neuronu biologicznego 22](#_Toc263023026)

[Rysunek 8. Schemat sztucznego neuronu 22](#_Toc263023027)

[Rysunek 9. Wykres funkcji y = tanh(ßx) 24](#_Toc263023028)

[Rysunek 10. Sieć MLP z jedną warstwą ukrytą 25](#_Toc263023029)

[Rysunek 11. Sieć MLP z wieloma warstwami ukrytymi 26](#_Toc263023030)

[Rysunek 12. Schemat sieci Hopfielda 27](#_Toc263023031)

[Rysunek 13. Diagram komponentów biblioteki CNL oraz jej otoczenia 35](#_Toc263023032)

[Rysunek 14. Diagram klas biblioteki CNL (część pierwsza) 36](#_Toc263023033)

[Rysunek 15. Diagram klas biblioteki CNL (część druga) 37](#_Toc263023034)

[Rysunek 16. Diagram sekwencji uczenia sieci w bibliotece CNL 38](#_Toc263023035)

[Rysunek 17. Diagram sekwencji uruchamiania sieci na CPU 43](#_Toc263023036)

[Rysunek 18. Diagram sekwencji trenowania sieci na CPU 44](#_Toc263023037)

[Rysunek 19. Diagram sekwencji uruchamiania sieci na GPU 45](#_Toc263023038)

[Rysunek 20. Diagram sekwencji trenowania sieci na GPU 46](#_Toc263023039)

[Rysunek 21. Program CUDA Visual Profiler 60](#_Toc263023040)

# Spis tabel

Tabela 1. Porównanie technologii GPGPU 12

Tabela 2. Formaty plików danych obsługiwane przez bibliotekę CNL 39

Tabela 3. Lista katalogów i plików w projekcie razem z opisami 40

Tabela 4. Opis użytych kerneli 48

Tabela 5. Jakość uczenia sieci w zależności od różnych parametrów 54

Tabela 6. Porównanie czasu uruchomienia sieci na CPU i GPU 56

Tabela 7. Porównanie czasu uczenia sieci na platformach 32 i 64-bitowych 57

1. Nazywanych w skrócie GPU (ang. graphics processing unit). [↑](#footnote-ref-1)
2. Operacje wykonywane na procesorze CPU (w odróżnieniu od tych wykonywanych na GPU) są też nazywane operacjami wykonywanymi na hoście. [↑](#footnote-ref-2)
3. Ang. Compute Unified Device Architecture. [↑](#footnote-ref-3)
4. Ang. multi-layer perceptron – jeden z najpopularniejszych typów sieci neuronowych. Został opisany w rozdziale . [↑](#footnote-ref-4)
5. Ang. general purpose computing on graphics processing units – technika użycia procesora graficznego GPU do ogólnych zastosowań. [↑](#footnote-ref-5)
6. Do tej pory operacje 3D musiały być obliczane przez główny procesor komputera. [↑](#footnote-ref-6)
7. Porównując prędkości GPU oraz CPU z 1999 roku z tymi z 2009 roku: według , karta GeForce 256 posiadała zdolność wypełniania 480 mln tekseli/s, karta GeForce GTX 285 – ponad 50 mld tekseli/s - różnica ponad 100 razy. Wg. , procesor Pentium III 1000Mhz ma 2205 pkt. w teście, natomiast procesor Core i7 2800Mhz ma 10094 pkt., ma też 4 rdzenie, więc różnica prędkości to 18.3, dużo mniejsza niż w przypadku GPU. [↑](#footnote-ref-7)
8. Ang. floating point operations per second *–* liczba operacji zmiennoprzecinkowych na sekundę. [↑](#footnote-ref-8)
9. Ang. Arithmetic Logic Unit, układ wykonujący operacje na liczbach całkowitych. [↑](#footnote-ref-9)
10. Ang. Floating Point Unit. [↑](#footnote-ref-10)
11. Nie ma konieczności używania wielu instrukcji sterujących for, do, while. [↑](#footnote-ref-11)
12. Ang. streaming multiprocessor (SM). [↑](#footnote-ref-12)
13. Ang. scalar processor (SP). [↑](#footnote-ref-13)
14. Compute capability (lub CC) określa architekturę jądra GPU. Jest to dokładniej opisane w rozdziale . [↑](#footnote-ref-14)
15. Technologia Hardware T&L (ang. Tranform and Lighting). [↑](#footnote-ref-15)
16. Ang. Software Development Kit. [↑](#footnote-ref-16)
17. Więcej informacji na temat składni programów CUDA jest w  . [↑](#footnote-ref-17)
18. Dokumentacja zaleca, by była przynajmniej dwa razy większa od ilości multiprocesorów. [↑](#footnote-ref-18)
19. W nowej architekturze Fermi nie ma już takich różnic. [↑](#footnote-ref-19)
20. Ang. Error Correction Code – kody korekcyjne. [↑](#footnote-ref-20)
21. Ze wzoru w źródle została usunięta zmienna t oraz poprawiony został wzór 3.5 – we wzorze w książce, wszystkie warstwy muszą mieć tę samą funkcję aktywacji. [↑](#footnote-ref-21)
22. Ang. Support Vector Machine, opisana w (Cristianini, 2007). [↑](#footnote-ref-22)
23. Ang. Extensible Markup Language. [↑](#footnote-ref-23)
24. Ang. Comma Separated Values. [↑](#footnote-ref-24)
25. Opisane są one dokładniej w rozdziale . [↑](#footnote-ref-25)
26. Wymagałoby to zakomentowania linii „#define REAL\_GPU\_IS\_FLOAT 1” wewnątrz pliku stdafx.h oraz zmienienia docelowej architektury pliku TrainNetwork.cu na sm\_13. [↑](#footnote-ref-26)
27. Ang. Unified Modeling Language. [↑](#footnote-ref-27)
28. Jest on dokładniej opisany w rozdziale i . [↑](#footnote-ref-28)
29. Kernele to funkcje wykonywane na GPU, opisane dokładniej w rozdziale . [↑](#footnote-ref-29)
30. Jest to dokładniej opisane w rozdziale . [↑](#footnote-ref-30)
31. Szerzej jest ona opisana w rozdziale . [↑](#footnote-ref-31)
32. Oznacza to duże sieci neuronowe, zestawy testów z dużą ilością wejść i wyjść, duża ilość testów użytych w trenowaniu w każdej iteracji. [↑](#footnote-ref-32)
33. Synchronizacja między wątkami jest dokładniej opisana w rozdziale . [↑](#footnote-ref-33)
34. Wszystkie testy przy których nie zaznaczono inaczej, były uruchamiane w systemie Vista 32bit. [↑](#footnote-ref-34)
35. W marcu 2010 udostępniono framework CUDA w wersji 3.0, jednak nie jest ona dopracowana i w wielu sytuacjach programy w nim uruchomione działają wolniej niż w wersji 2.3 – więcej informacji na  i . [↑](#footnote-ref-35)
36. Wyniki w tym przypadku są błędne. Może to być spowodowane błędnie działającą biblioteką CUDA Toolkit lub sterownikiem graficznym. [↑](#footnote-ref-36)
37. DOM (ang. Document Object Model), opisany w (W3C, 2009)). [↑](#footnote-ref-37)
38. Nie dotyczy to platformy Fermi – tam spowolniłoby to działanie – zobacz . [↑](#footnote-ref-38)