Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Informatyki, Elektroniki i Telekomunikacji

KATEDRA INFORMATYKI

ROZPRAWA DOKTORSKA

DOMINIK ŻUREK

AKCELERACJA OBLICZEŃ ALGORYTMÓW UCZENIA MASZYNOWEGO ORAZ WYBRANYCH POPULACYJNYCH ALGORYTMÓW INTELIGENCJI OBLICZENIOWEJ ZE ZREDUKOWANĄ PRECYZJĄ DANYCH POPRZEZ IMPLEMENTACJĘ W UKŁADACH GPGPU

PROMOTOR:
prof. dr hab. inż. Kazimierz Wiatr

PROMOTOR POMOCNICZY:
dr. inż. Marcin Pietroń

Kraków 2020

OŚWIADCZENIE AUTORA PRACY

OŚWIADCZAM, ŚWIADOMY ODPOWIEDZIALNOŚCI KARNEJ ZA POŚWIADCZENIE NIEPRAWDY, ŻE NINIEJSZĄ PRACĘ DYPLOMOWĄ WYKONAŁEM OSOBIŚCIE I SAMODZIELNIE, I NIE KORZYSTAŁEM ZE ŹRÓDEŁ INNYCH NIŻ WYMIENIONE W PRACY.

. . . . . .

PODPIS

AGH

University of Science and Technology in Krakow

Faculty of Computer Science, Electronics and Telecommunications

DEPARTMENT OF COMPUTER SCIENCE

PHD OF SCIENCE THESIS

DOMINIK ŻUREK

ACCELERATING MACHINE LEARNING ALGORITHMS AND SELECTED POPULATION COMPUTATIONAL INTELLIGENCE ALGORITHMS OF ARBITRARY-PRECISION ARITHMETICS ON THE GPGPU

SUPERVISOR:
prof. dr hab. inż. Kazimierz Wiatr

CO-SUPERVISOR:
dr. inż. Marcin Pietroń

Krakow 2020

Pragnę podziękować Panu Profesorowi Kazimierzowi Wiatrowi oraz Panu Doktorowi Marcinowi Pietroniowi za cenne rady, zaangażowanie oraz słowa mobilizacji podczas pisania tej pracy.

Spis treści

1. Wstęp

1.1. Motywacja

W ostatnich latach można zaobserwować ogromny wzrost liczby generowanych przez ludzkość danych. Jak podają różnego rodzaju statystki liczba ta sięga setek trylionów bajtów każdego dnia. Szacuje się, że obecnie każdy człowiek w czasie jednej sekundy, generuje prawie 2 [MB] danych. Najwięcej z nich zostaje wygenerowanych przy okazji wyszukiwania treści w internecie jak również przez korzystanie z szeroko rozwiniętych komunikatorów oraz portali społecznościowych. Głównym generatorem danych są coraz to lepiej rozwinięte i coraz szerzej dostępne urządzenia mobilne, w których od paru lat dostęp do internatu z każdego miejsca stał się standardem. Zwiększająca się popularność tak wyposażonych urządzeń mobilnych powoduje że liczba wyprodukowanych przez ludzkość danych z każdym rokiem będzie się systematycznie zwiększać. Rodzi to potrzebę ciągłego rozwoju mocy obliczeniowej, dzięki której możliwe staje się przetworzenie tak kolosalnej ilości informacji.

Na przestrzeni ostatnich lat popularnymi akceleratorami, ze względu na swoją efektywność, stały się karty graficzne z procesorami GPU. Przez długi okres wykorzystanie kart graficznych ograniczało się do generowania obrazów wraz z wykonywaniem na nich odpowiednich algorytmów. Wraz z potrzebą wytwarzania obrazów coraz to wyższej jakości, urządzenia te były stopniowo wyposażane w coraz to większą moc obliczeniową, co wzbudziło zainteresowanie akceleratorami tego typu, w celu użycia ich do rozwiązań zarówno naukowych jak i komercyjnych. Obecnie karty graficzne znajdują swoje zastosowanie wszędzie tam, gdzie możliwe jest równoległe przetwarzanie danych, więc tak naprawdę w każdej dziedzinie naukowej jak również komercyjnej. w tym czasie zmodyfikowane także architektury procesorów GPU w kierunku ich większej elastyczności - stąd nazwa GPGPU. Akceleratory graficzne mogą być używane do rozwiązywania problemów wysoko specjalizowanych czyli samodzielnie przetwarzać silnie równoległy algorytm, gdzie z racji ich architektury uzyskuje się największe przyśpieszenia, bądź mogą wykonywać tylko najbardziej krytyczny, dający się zrównoleglić, fragment algorytmu (nazywany potocznie *wąskim gardłem*), czyli mogą być częścią systemu hybrydowego. Szczególnie w tym drugim przypadku należy mieć na uwadze, główny problem pojawiający się przy okazji stosowania akceleratorów sprzętowych tj. transmisję danych pomiędzy procesorem a akceleratorem. Innym problemem, napotykanym podczas programowania różnego rodzaju akceleratorów jest rozmiar oraz szybkość dostępu do pamięci. Oba te problemy są sukcesywnie rozwiązywane poprzez nieustanne wprowadzanie na rynek, coraz to nowocześniejszych wersji akceleratorów (co jest szczególnie widoczne w przypadku kart graficznych), dając tym samym coraz to szersze spektrum ich zastosowań.

Ważnym aspektem w kontekście najnowszych akceleratorów graficznych jest reprezentacja danych, na jakich zostanie przeprocesowany dany algorytm. Większość obliczeń wykorzystuje format zmiennoprzecinkowy do którego zapisu używany jest standard IEEE-754, gdzie najczęściej do reprezentacji liczby używa się pojedynczą (*float*) bądź podwoją (*double*) precyzję. Oczywistością jest, że wyższa precyzja do zapisu liczby wymaga wykorzystania większej liczby bitów, a co za tym idzie potrzebuje zużyć więcej pamięci. Problem dostępnej pamięci może stanowić barierę do efektywnej implementacji danego algorytmu w akceleratorze sprzętowym. Z tego też względu najnowsze modele procesorów graficznych, wprowadziły możliwość reprezentacji danych przy użyciu zredukowanej precyzji, co oznacza możliwość zapisu liczby na mniejszej liczbie bitów. Co więcej, wprowadzano osiągające wysokie przyśpieszenia, specjalistyczne rdzenie zwane *Tensor core*, do których użycia wymagana jest reprezentacja liczb wykorzystująca 16 lub 8 bitów. Należy mieć na uwadze, że oprócz ogromnych korzyści związanych z zastosowaniem zredukowanej precyzji tj. większej szybkości obliczeń jak również oszczędności pamięci, użycie mniejszej liczby bitów do reprezentacji liczby, prowadzi do przechowywania mniejszej ilości informacji, co w konsekwencji może skutkować spadkiem skuteczności algorytmu. Aby możliwe było wykorzystanie profitów związanych z użyciem zredukowanej precyzji danych nie tracąc przy tym najważniejszego aspektu jakim jest efektywność algorytmu, wprowadza się proces konwersji danych z ich oryginalnej reprezentacji do zredukowanej formy zwany *kwantyzacją*. Dobór odpowiedniego algorytmu kwantyzacji niejednokrotnie pozwala na przeprowadzenie obliczeń na liczbach zapisanych w ich zredukowanej formie zachowując jednocześnie niezawodność algorytmu.

Jedną z grup algorytmów, dla których kluczową rolę odgrywa akceleracja obliczeń są algorytmy ewolucyjne, używane do rozwiązywania problemów, których nie da się opisać przy pomocy dokładnych formuł, bądź liczba kombinacji (punktów) jakie należy sprawdzić by otrzymać rozwiązanie jest zbyt duża by użyć w tym celu prostego algorytmu. W problemach tego typu, aby możliwe było znalezienie rozwiązania, koniecznie staje się wprowadzenie technik heurystycznych, podejmujących decyzję o przeszukiwaniu przestrzeni na podstawie dotychczas uzyskanych rozwiań oraz zależności pomiędzy nimi. Podejście takie nie daje gwarancji uzyskania optymalnego rozwiązania, jednak pozwala na szybkie znalezienie rozwiązań takich, dla których cel optymalizacyjny zostaje spełniony. W procesie poszukiwań satysfakcjonujących rozwiązań ważnym aspektem jest możliwość jak najszerszej eksploracji przestrzeni w określonym czasie. Właśnie w tym elemencie z pomocą przychodzą akceleratory sprzętowe, dzięki którym w tym samym czasie możliwe staje się przeszukanie większego obszaru, co zwiększa szanse na znalezienie bardziej optymalnego rozwiązania. Złożone systemy wyposażone w inteligentne techniki obliczeniowe, tworzone dla potrzeb przetwarzania ogromnych zbiorów danych czy też rozwiązywania trudnych obliczeniowo problemów, można spotkać w takich dziedzinach jak medycyna, przemysł, ekonomia czy też nauka, co pokazuje jak szerokie spectrum zastosowań mają algorytmy ewolucyjne. Takie systemy z racji ich skomplikowania są najczęściej realizowane w formie hybrydowej, co daje możliwość wprowadzenie akceleratorów odpowiedzialnych za przetworzenie najbardziej newralgicznych części algorytmu.

Kolejną grupą algorytmów, potrzebującą do poprawnego działania ogromnej ilości danych, a co za tym idzie dostępu do mocy obliczeniowej, są algorytmy uczenia maszynowego. Metody uczenia maszynowego wykorzystują dorobek informatyki, statystyki oraz teorii rozpoznawania struktur umożli-

wiając automatyzację akwizycji wiedzy co stanowi alternatywę dla tworzenia reguł przez ekspertów w danej dziedzinie. Wspomniana wcześniej nieustannie rosnąca ilość przetwarzanych danych prowadzi do konieczności rozwijania narzędzi umożliwiających automatyczną ich analizę. Można powiedzieć, że tempo przyrostu danych chociażby w sieciach komputerowych jest wykładnicze, w związku z tym istnieje pilna potrzeba ich bardzo szybkiego przetwarzania (akceleracja). Najlepsze osiągnięcia akceleracyjne uzyskuje się dzięki przetwarzaniu danych w sposób równoległy, gdzie idealnym sposobem jest wykorzystanie układów GPGPU, dzięki którym możliwe jest uzyskanie bardzo wysokiej wydajności obliczeniowej systemu. Należy podkreślić, że systemy takie do poprawnego działania potrzebują przechowywać ogromną ilość parametrów, więc w tym przypadku bardzo ważną rolę odgrywa wyżej wspominany proces kwantyzacji, który przyśpiesza obliczenia (akceleracja) oraz pozwala zaoszczędzić pamięć (która w przypadku akceleratorów sprzętowych jest bardzo ograniczona), istotnie zwiększając w ten sposób liczbę wykorzystywanych próbek uczących (zwiększa dokładność systemu). Obecnie systemy wykorzystujące algorytmy uczenia maszynowego można spotkać w wielu dziedzinach ludzkiego życia. Zadaniem takich systemów jest uczenie się naszego zachowania i na tej podstawie generacja wyników, poszukując tym samym zależności i regularności naszych poczynań. Łatwo zauważyć jak szeroka jest grupa zastosowań dla aplikacji tego typu. W związku z tym powstaje pilna potrzeba ciągłego ulepszania całej gamy algorytmów, poprzesz tworzenie ich bardziej skutecznych oraz szybszych obliczeniowo wersji.

1.2. Cel i teza pracy

Od wielu lat trwają badania nad przyśpieszaniem obliczeń przy użyciu różnego rodzaju akceleratorów sprzętowych [17][82][83][119][121][122][129]. Szczególne znaczenie mają one w systemach korzystających z różnego rodzaju algorytmów sztucznej inteligencji, które do realizacji swoich celów wymagają przetworzenia ogromnej liczby danych wejściowych oraz wymagają dostępu do dużej mocy obliczeniowej. Niejednokrotnie w systemach tego typu wymagana jest możliwość podejmowania przez nie decyzji w czasie rzeczywistym (np. autonomiczny samochód), stąd też bardzo pożądane jest opracowywanie nowych sposób umożliwiających szybsze procesowanie takich algorytmów.

Akceleratorami sprzętowymi osiągającymi najlepsze rezultaty w kontekście algorytmów związanych ze sztuczną inteligencją są układy GPGPU. Każdego roku producenci wprowadzają na rynek coraz to mocniejsze procesory graficzne często wyposażone w nowe funkcjonalności co powoduje konieczność dopasowywania istniejących implementacji do nowych architektur w celu osiągnięcia coraz to lepszych wyników. Ostatnią funkcjonalnością będącą krokiem milowym w osiągnięciach akceleratorów tego typu, było wprowadzenie możliwości używania zredukowanej precyzji danych co owocuje szybszymi obliczeniami oraz oszczędnością pamięci. Nadało to większe znaczenie algorytmom kwantyzacji, które polegają na takiej zamianie liczby z jej oryginalnej postaci do zredukowanej formy, by nie stracić efektywności algorytmu.

Celem pracy jest opracowanie i zbadanie skutecznych i efektywnych metod przyśpieszania obliczeń związanych z algorytmami sztucznej inteligencji, nadających się do przetwarzania równoległego, poprzez ich implementację w układach GPGPU. Praca przedstawia możliwości wykorzystania tych ak-

celeratorów w problemach związanych z inteligencją obliczeniową jak również z algorytmami uczenia maszynowego. Zawarte w niej implementacje pokazują możliwości kart graficznych oraz zawierają porównania implementacji tych samych algorytmów przy użyciu różnych szerokości danych. Dodatkowo zostały przedstawione techniki kwantyzacji pozwalające zredukować precyzję zapisu danych, pokazując przy tam jak bardzo można ograniczyć reprezentację by nie spowodować znaczącego spadku skuteczności algorytmu.

W wyniku powyższych założeń sformułowano następującą tezę:
„Implementacja algorytmów inteligencji obliczeniowej oraz uczenia maszynowego w akceleratorach GPGPU prowadzi do przyśpieszenia ich wykonania w stosunku do implementacji CPU. Użycie w implementacji sprzętowej zredukowanej precyzji danych, poprawia szybkość kart graficznych, nie zawsze prowadząc do pogorszenia jakości algorytmu”.

1.3. Organizacja pracy

Niniejszą pracę podzielono na 7 rozdziałów poświęconych zagadnieniom implementacji algorytmów inteligencji obliczeniowej oraz uczenia maszynowego układach GPGPU jak również problematyce efektywnej kwantyzacji oraz bibliografii.

Rozdział 1: „Wstęp” stanowi wprowadzenie do poruszanej w pracy tematyki. Rozdział zawiera tezę pracy i jej główny cel. Dodatkowo poruszane są w nim informacje na temat motywacji do podjęcia takiej tematyki.

Rozdział 2: „Sztuczna inteligencja, inteligencja obliczeniowa, uczenie maszynowe i głębokie uczenie maszynowe” zawiera opis teoretyczny algorytmów sztucznej inteligencji. Główny nacisk został położony na dziedziny, z których wywodzą się algorytmy, objęte próbą efektywnej implementacji w niniejszej pracy (ewolucyjne algorytmy agentowe, sieci neuronowe).

Rozdział 3: „Platformy sprzętowe” przestawia architekturę współczesnych układów GPGPU, jako akceleratorów, które zostały użyte na potrzeby niniejszej pracy. Opisuje charakterystykę konkretnego modelu, użytego w tej pracy (Nvidia Tesla V100-SXM2-32GB [96]). W rozdziale tym znalazł się również opis narzędzia umożliwiającego implementację algorytmów w procesorach graficznych (CUDA [110]).

Rozdział 4: „Równoległa implementacja algorytmów optymalizacji problemu *Low autocorrelation binary sequence* w układach GPGPU” opisuje dokładnie problem *LABS* wraz z algorytmami *SLDS*, *TABU*, *SLDS-2* oraz *SLDS-przeszukiwanie w głąb* przy pomocy, których zostały podjęte próby jego rozwiązania. Na początku rozdziału zostaje przedstawiona koncepcja algorytmów heurystycznych. Najważniejszą częścią rozdziału są opisy równoległej implementacji wspominanych algorytmów na akceleratorach graficznych.

Rozdział 5: „Trenowanie algorytmu wektorów nośnych ze zredukowaną precyzją danych w celu klasyfikacji tekstu". Rozdział ten zawiera opis najnowszych modeli używanych do przetwarzania języka naturalnego, opis algorytmu wektorów nośnych, opisuje format zmiennoprzecinkowy liczby oraz przybliża dwie metody kwantyzacji użyte w tej pracy. Nadrzędnym celem rozdziału jest pokazanie wpływu dwóch zaproponowanych metod kwantyzacji w procesie trenowania wektorów nośnych na skuteczność algorytmu użytego w celu klasyfikacji tekstu. Dodatkowo pokazano wpływ użycia zredukowanej precyzji danych na szybkość wykonania algorytmu w układach GPGPU.

Rozdział 6: „Efektywne sposoby obliczania operacji konwolucji przy użyciu układów GPGPU". W rozdziale tym zostały podjęte próby zaimplementowania efektywniejszego sposobu obliczania operacji splotowych na procesorach graficznych od tych oferowanych przez bibliotekę *cuDnn* [99]. Na początku rozdziału zostały przestawione motywacje do podjęcia takich działań, po czym znajduje się opis modeli wykorzystanych do testowania zaproponowanego podejścia. W rozdziale zostały zawarte dokładne opisy różnych metod obliczania operacji splotowych oraz dokładny opis wraz z implementacją nowego algorytmu obliczania konwolucji nazwanego *konwolucją rzadką*. Ważnym elementem rozdziału jest opis techik pozwalających na zredukowanie objętości sieci (*Pruning*) oraz wpływ użycia zredukowanej precyzji na szybkość przetwarzania warstw konwolucyjnych zarówno przez *cuDnn* jak również metodę *konwolucji rzadkiej* Podsumowaniem rozdziału jest informacja o tym przy jakich warunkach bardziej opłacalne jest wykorzystanie w obleczeniach konwolucji przedstawionej metody *konwolucji rzadkiej* zamiast tych zaimplementowanych w bibliotece *cuDnn*.

Rozdział 7: Podsumowanie to rozdział zawierający wnioski oraz uwagi końcowe".

Bibliografia: zawiera 129 pozycji, z których skorzystano podczas tworzenia niniejszej rozprawy.

2. Sztuczna inteligencja, inteligencja obliczeniowa, uczenie maszynowe i głębokie uczenie maszynowe

Pojęciu sztucznej inteligencji (ang. artificial intelligence - AI) przypisuje się kilka znaczeń, najczęściej jednak nazwa ta definiowana jest jako: **uczące się maszyny lub systemy informatyczne, które przetwarzają informację w oparciu o reguły ludzkiego rozumowania.** Sztuczna inteligencja zajmuje się zagadnieniami, które nie są efektywnie algorytmizowane w oparciu o modelowanie wiedzy, więc do ich rozwiązania należy wprowadzić algorytmy posiadające znamiona inteligencji. Przez stwierdzenie inteligencji rozumie się, zdolność do samodzielnego przystosowania się do zmiennych warunków, podobnie jak ma to miejsce w przypadku ludzkiej inteligencji, która jest wzorem działania dla AI. Można więc powiedzieć, że AI jest próbą przeniesienia właściwości ludzkiego mózgu na programy uruchamiane na maszynach (komputerach), więc ma za zadanie uczyć maszyny zachowań podobnych do tych ludzkich. Przez proces uczenia się systemu rozumie się dokonanie autonomicznej zmiany w systemie, zachodzącej na podstawie odbytych doświadczeń i prowadzącej do poprawy jakości działania. Przy takiej definicji, zakłada się, że istnieje możliwość oceny jakości podejmowanych decyzji, czyli umiejętność odróżnienia zmian korzystnych od niekorzystnych. Sztuczna inteligencja potrafi przyswajać wiedzę poprzez wyodrębnianie wzorców z surowych danych. Wprowadzając do algorytmu elementy ludzkiej inteligencji, możliwe jest wytrenowanie go tak by potrafił rozpoznawać obrazy, rozumieć język naturalny czy też by był zdolny do logicznego rozumowania. Warto podkreślić, że funkcjonuje rozróżnienie AI na dwie grupy:

– Weak AI- zwana również *narrow AI*, skupia się na wąskim zdefiniowanym z góry zadaniu i nie wychodzi poza jego obszary

– Strong AI - zwana również *general AI*, ma na celu naśladowanie pełnego zakresu ludzkich możliwości poznawczych, czyli obejmuje system z wszechstronną wiedzą i zdolnościami poznawczymi.

Jedną z poddziedzin sztucznej inteligencji jest tzw. **inteligencja obliczeniowa** (ang. *computational Intelligence - CI*), która głównie polega na zdolności przystosowania się systemu do zmieniającego się środowiska, pokładającą duży nacisk na ulepszanie i rozwój aplikacji w świecie rzeczywistym. Odróżniającą cechą algorytmy tego typu od innych algorytmów sztucznej inteligencji jest brak korzystania z góry zdefiniowanego modelu, lecz podejmowanie prób zbudowania go samodzielnie na podstawie dostarczonych zbiorów uczących. Algorytmy inteligencji obliczeniowej obejmują algorytmy wywodzące się ze sztucznej inteligencji związane z inteligentnym przetwarzaniem danych, z których istotną grupę

stanowią algorytmy ewolucyjne, których inspiracją są procesy naturalnej ewolucji organizmów, żyjących w pewnym środowisku, potrafiącym utrzymać określoną liczbę osobników posiadających zdolność do rozmnażania się oraz u których występuję zjawisko śmiertelności, stanowiące formę selekcji. Osobniki najlepiej przystosowane do środowiska (czyli te które najlepiej rywalizują o zasoby), mają największe szanse na rozmnażanie się. U potomstwa zwykle występują niewielkie zmiany tzw. mutacje, które prowadzą do osiągnięcia lepszej zdolności dopasowywania się do środowiska. Osobniki, u których w wyniku mutacji odnotowano, słabe zdolności dopasowywania się do środowiska, mają małe szanse na przetrwanie, a co za tym idzie zimniejsza się ich szansa na wydanie potomków. W związku z tym wraz z upływem czasu, dobór naturalny sprawia, że ogólne przystosowanie osobników do środowiska rośnie. Przenosząc inspiracje biologiczne na algorytmy ewolucyjne, populacja osobników jest odzwierciedleniem liczby rozwiązań (osobnik = rozwiązanie). Ewolucja rozwiązań algorytmu, jest odpowiednikiem zmian występujących w populacji, za co odpowiedzialne są operatory mutacji (losowa modyfikacja) oraz rekombinacji (wymiana materiału genetycznego), co sprowadza się do wyszukiwania kolejnych rozwiązań. Pomiar dopasowania osobników do środowiska, czyli jakość reprezentowanych przez nich rozwiązań, odbywa się poprzez tzw. *funkcję oceny (ang. fitness function)*. Na podstawie tej funkcji odbywa się selekcja osobników najlepiej przystosowanych do środowiska, co ma prowadzić do generowania coraz to lepszych rozwiązań, dzięki czemu cały proces powinien zmierzać do znalezienia jak najbardziej optymalnego rozwiązania problemu. Poszukiwanie rozwiązań odbywa się w określonych ramach czasowych bądź przez ustaloną liczbę iteracji lub przerywane jest w momencie braku progresu (kolejne iteracje nie przynoszą lepszych rezultatów), co określane jest mianem warunku stopu algorytmu. Finalne rozwiązanie jest pobierane od najlepszego osobnika, który mógł zostać wygenerowany w dowolnej iteracji. Z ogólnego punktu widzenia algorytmy genetyczne można podzielić na:

– **algorytmy genetyczne** - za ich twórce uważa się Hollanda [46], dla którego inspirację stanowiła biologia. Algorytm genetyczny poszukując optymalnego wyniku, przeszukuje przestrzeń alternatywnych rozwiązań problemu, gdzie momentem startu staje się pewna populacja punktów startowych, a nie pojedynczy punkt. Podczas poszukiwań optymalnych rozwiązań, parametry zadania przetwarzanie są przy użyciu zakodowanej formy w postaci chromosomu (ciąg kodowy stanowiący uporządkowany ciąg genów czyli cech lub znaków). Wykorzystywane są przede wszystkim w problemach optymalizacyjnych, stosując do ich rozwiązania metody probabilistyczne a nie deterministyczne. W klasycznym algorytmie genetycznym spotkać można dwa podstawowe operatory generyczne tj. operator krzyżowania (ang. *crossover*) oraz operator mutacji (ang. *mutation*). Warty podkreślenia jest fakt, że to operator krzyżowania jest tym pierwszoplanowym, co ma przełożenie na częstość ich występowania. Operator krzyżowania odpowiada za za wymianę łańcuchów kodowych między dwoma osobnikami rodzicielskimi, na wyznaczonych w sposób losowy pozycjach zwanych *punktami krzyżowania*. Więcej na temat strategi doboru funkcji krzyżowania można znaleźć w [69]. Drugoplanowy operator tj. operator mutacji dokonuje zmiany wartości losowo wybranego bitu, co może nastąpić przed dokonaniem krzyżowania na populacji rodziców bądź na osobnikach utworzonych w wyniku krzyżowania tj. na osobnikach z populacji potomków. W algorytmach genetycznych wybór osobników do populacji, czyli selekcja, odbywa się poprzez

losowanie zależne od funkcji przystosowania. Do najpopularniejszych funkcji celu zalicza się selekcję proporcjonalną, rankingową oraz turniejową, które są dokładniej opisane w [35][69],

– **strategie ewolucyjne** - zostały zapoczątkowane przez Rechenberga [89] oraz Schwefela [92], jako metody służące do rozwiązywania trudnych problemów optymalizacji parametrycznej. Podobnie jak algorytmy genetyczne, metody te operują na populacjach potencjalnych rozwiązań oraz za pomocą selekcji wybierają te najlepiej przystosowane od środowiska. W tym przypadku jednak osobniki nie są reprezentowane poprzez wektory binarne, jak miało to miejsce w algorytmach genetycznych, tylko przez wektory liczb zmiennoprzecinkowych. Selekcja osobników jest w tym przypadku w pełni deterministyczna, co znaczy że osobniki nie mogą być powtórnie wybierane (w algorytmie genetycznym było to możliwe), co jest realizowane przez tworzenie populacji pośredniej, z której zawsze bezwarunkowo wybierane są osobniki najlepsze (w algorytmach genetycznych mogła wystąpić sytuacja, że osobniki najgorsze również były wybrane w procesie selekcji). Najważniejszym operatorem wariacji dla strategi ewolucyjnych jest mutacja. Dla reprezentacji zmiennoprzecinkowych odbywa się ona w niezależnym losowym zaburzeniu każdego elementu wektora. Drugorzędną rolę odgrywa rekombinacja, która jest realizowana przez rekombinację dyskretną, polegającej na wymianie poszczególnych parametrów między rodzicami, bądź przy pomocy operacji arytmetycznych, takich jak średnia arytmetyczna lub geometryczna, wykonanych na poszczególnych parametrach,

– **programowanie ewolucyjne** - zostało zaproponowane przez L.J Fogla [25] i obecnie jest wykorzystywane do rozwiązań optymalizacyjnych. W algorytmach tego typu nie występuję operator krzyżowania, gdyż, a głównym operatorem jest operator mutacji. Proces selekcji, podobnie jak w algorytmie genetycznym, jest probabilistyczny,

– **programowanie genetyczne** - jest to odmiana technik ewolucyjnych, rozwinięta przez J. Koza [57] i stanowi rozszerzenie klasycznego algorytmu genetycznego, którego idą jest automatyczna generacja programów komputerowych. Aby było to możliwe binarna reprezentacja została zastąpiona kodowaniem drzewiastym, w którego węzłach znajdują się wielkości niebinarne reprezentujące zmienne, symbole funkcji wywodzące się z pewnego alfabetu. Populacje programów rozwiązujących postawione przed nimi zadanie, poddawane są odpowiednim operatorom genetycznym, takim jak krzyżowanie oraz mutacji, oraz są opisane przez wartość funkcji oceny. Programy radzące sobie słabo z rozwiązaniem zadania są w procesie ewolucji eliminowane, natomiast te lepsze tworzą programy potomków, z których kolejne generacje są coraz to lepiej przystosowane do rozwiązania postawionego przed nimi problemu. Ponieważ występuje tu struktura drzewiasta, metody krzyżowania oraz mutacji dokonują wymiany w węzłach tych struktur.

Inną odmianą algorytmów ewolucyjnych są architektury agentowe, których fragment został zaimplementowany na potrzeby niniejszej pracy w układach GPGPU, dlatego też do ich opisu został przeznaczony osobny podrozdział 2.1.

Oprócz algorytmów ewolucyjnych do jednej z najważniejszych dziedzin sztucznej inteligencji zalicza się **uczenie maszynowe** (ang. *machine learning - ML*), które opiera się na algorytmach wyodrębnia-

jących informację z surowych danych, a następnie reprezentowaniu ich w formie określonego modelu, zdolnego do przetwarzania kolejnych nie za-modelowanych danych czyli np. do wyszukiwania wzorców, przewidywania rezultatów bądź podejmowania decyzji. Uczenie maszynowe można podzielić na cztery główne rodzaje:

1. **uczenie nadzorowane** (ang. *supervised learning*) - model uczy się przypisywania opisanych wcześniej przez człowieka etykiet do danych wejściowych. Inaczej mówiąc, w uczeniu z nadzorem do modelu dostarczane są dane, zawierające oczekiwaną odpowiedź. Mogą to być zdjęcia z zaznaczonym fragmentem, który będzie poszukiwany (np. oczy), bądź zestaw e-maili z informacją, które z nich są spamami. Dzięki takiemu podejściu, system jest zdolny do wyszukania wzorca na zdjęciu, które nie było częścią zbioru treningowego, bądź jest w stanie odpowiedzieć na pytanie czy przychodząca wiadomość jest spamem. Do grupy tych algorytmów zaliczają się między innymi: drzewa decyzyjne [90], metoda najbliższych sąsiadów [37], regresja liniowa [8], naiwny algorytm Bayesa [128], jak również opisane w późniejszych rozdziałach metoda wektorów wspierających oraz sieci neuronowe,

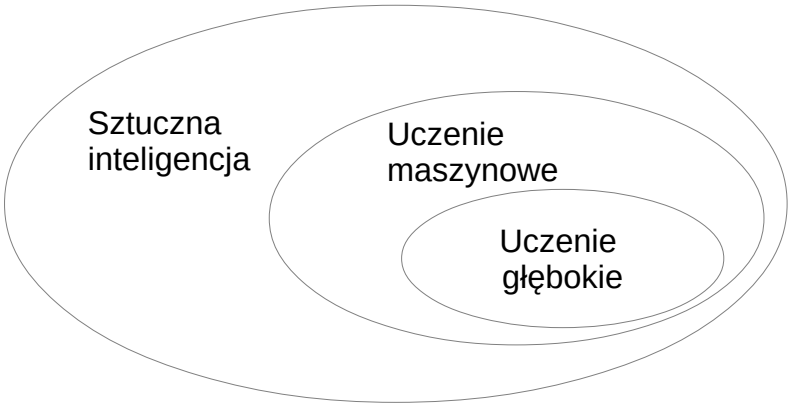
2. **uczenie nienadzorowane** (ang. *unsupervised learning*) - model zajmuje się określaniem transformacji danych wejściowych bez pomocy docelowych etykiet. Uczenie takie głównie opiera się na poznawaniu korelacji między danymi, co potem jest wykorzystywane w zadaniach związanych z wizualizacją danych, ich kompresją, pozbywaniem się z nich szumu, czy też segmentacją podobnych obiektów. Uczenie nienadzorowane zakłada brak informacji o oczekiwanym wyjściu, więc poszukiwanie zależności między danymi odbywa się całkowicie bez udziału człowieka. Algorytmy tego typu znajdują zastosowanie w poszukiwaniu nowych nieznanych dotąd regularności w próbce danych, co z kolei może okazać się przydatne przy wyszukiwaniu anomalii. Jednym z bardziej popularnych algorytmów nienadzorowanych jest algorytm k-means [22].

3. **uczenie częściowo nadzorowane** (ang. *semi-supervised learning*) - jest specyficznym rodzajem uczenia nadzorowanego. W tym przypadku nie występują dane z etykietami, dostarczane przez człowieka. Zamiast tego etykiety są generowane automatycznie przez algorytm, na podstawie danych wejściowych. W takich przypadkach etykiety są niezmodyfikowanymi danymi wejściowymi. Przykładem takich algorytmów są autokodery [127] oraz opisane w kolejnych rozdziałach, algorytmy służące do budowania języka naturalnego (rozdział 5.1.4).

4. **uczenie przez wzmacnianie** (ang. *reinforcement learning*) - działa w środowisku zupełnie nie znanym, brak jest określonych danych wejściowych i wyjściowych. Maszyna ucząca dostaje tzw. sygnał wzmocnienia, który może być pozytywny (nagroda) bądź negatywy (kara). Metodę tę można inaczej nazwać metodą prób i błędów. Przykładem może być gra w nową grę, nie znając reguł. W takim przypadku dopiero po skończonej grze pojawia się informacja o zwycięstwie (nagrodzie) bądź porażce (karze). Za zbieranie informacji ze środowiska odpowiedzialny jest tzw. **agent**. Na podstawie zebranych informacji agent podejmuje akcje mające na celu zmaksymalizowanie nagrody. Algorytmy te są używane przede wszystkim w grach takich jak szachy czy go. Popular-

nymi algorytmami używanymi w uczeniu przez wzmacnianie są : Q-learning, Temporal Difference (TD), Deep Adversarial Networks, których dokładny opis znajduje się w [115].

Najważniejszą dziedziną uczenia maszynowego jest **głębokie uczenie maszynowe** (ang. *deep learning -DL*). Przez głębokie rozumie się tworzenie wielowarstwowych reprezentacji danych. Głębokość modelu, określa liczba warstw jakie posiada. Przyjmuje się, że o głębokim uczeniu można mówić, gdy model posiada więcej niż trzy warstwy. Nie istnieje jednak formalna definicja określająca, kiedy mamy do czynienia z uczeniem płytkim czy głębokim, a wspominania liczba warstw ma charakter umowny. Zależności pomiędzy sztuczną inteligencją, uczeniem maszynowym, a głębokim uczeniem maszynowym doskonale obrazuje rysunek 2.1 W przypadku uczenia głębokiego wielowarstwowe reprezentacje danych



Rysunek 2.1. Relacje pomiędzy sztuczną inteligencją, uczeniem maszynowym, a głębokim uczeniem maszynowym

są w większości przypadków kojarzone z modelami określanymi mianem **sieci neuronowych**, których działanie oraz główne typy, które w jakiś sposób zostały użyte w niniejszej pracy, zostaną opisane w kolejnych podrozdziałach teoretycznych, co ułatwi późniejsze opisy praktyczne.

2.1. Ewolucyjne algorytmy agentowe

Mówiąc o ewolucyjnych systemach agentowych należy rozpocząć od definicji samego agenta, która została zaczerpnięta z [26], a która przedstawia się następująco : „agent to system, który jest usytuowany w pewnym środowisku i którego jednocześnie jest częścią; agent obserwuje (odbiera, odczuwa) to środowisko oraz działa w nim, w czasie, według własnego planu, wpływając na to, co będzie mógł zaobserwować w przyszłości". Typowo za agenta uznaje się autonomicznie podejmujący działania program komputerowy, uruchomiony w określonym środowisku, korzystający z jego zasobów, mający na celu rozwiązanie postawionych przed nim zadań. Do najistotniejszych cech agenta należy zaliczyć [52]:

– **autonomiczność** - najprościej rzecz ujmując cecha ta definiuje samodzielność agenta w podejmowaniu przez niego akcji oraz sprawowanie przez niego całkowitej kontroli nad swoim stanem wewnętrznym,

– **usytuowanie** - środowisko w jakim agent się znajduje, z którego zasobów korzysta i na które poprzez podejmowane przez siebie działania wpływa,

– **reaktywność** - cecha pozwalająca na zidentyfikowanie swojego środowiska i reagowanie na zmiany w nim zachodzące,

– **działania docelowe** - realizacja celów postawionych przed agentem poprzez podejmowanie odpowiednich akcji, mający wpływ na przyszłe obserwacje,

– **umiejętność uczenia się** - zdolność agenta do dostosowania się do zmiennych warunków środowiska,

– **zdolności socjalne** - agenci są zdolni do współpracy między sobą w celu zrealizowania zadania,

– **mobilność** - agent potrafi poruszać się w środowisku w którym został umiejscowiony,

– **racjonalność** - agent nie podejmuje działań, które mogły by być sprzeczne z wyznaczonym dla niego celem.

Tak zdefiniowany agent stanowi fundament koncepcji *agentowych obliczeń ewolucyjnych*. Główną motywacją do wprowadzenia tego typu algorytmów była chęć zdecentralizowania obliczeń ewolucyjnych, gdyż w procesie ewolucji do realizacji procesu selekcji oraz generacji nowych pokoleń wykorzystywany jest jeden algorytm, posiadający wiedzę o całej populacji [9]. Z tego też względu systemy te są pozbawione możliwości podejmowania samodzielnych decyzji czy też pobierania informacji z otaczającego ich środowiska, co wprowadza spore w ich działaniu ograniczenia. Z tego też względu wprowadza się do populacji agentów mechanizmy ewolucyjne, co przez niezależność w podejmowaniu przez agentów decyzji oraz ich wzajemnie oddziaływania, prowadzi do zdecentralizowania procesu ewolucji, który prowadzony jest przez populację agentów, które są odpowiednikiem pojedynczych osobników w algorytmie ewolucyjnym. Agenci wchodzący w skład systemu, mogą realizować swoje cele, które nie zawsze muszą być zgodne z interesem całej grupy oraz mogą mieć różne poziomy autonomii. W tak zdefiniowanym systemie, o zdecentralizowanej naturze, gdzie decyzje są podejmowane w sposób autonomiczny przez agentów (nie ma możliwości bezpośredniego wymuszenia na agencie podjęcia konkretnego działania), nie ma możliwości wykorzystania scentralizowanych mechanizmów sterowania przebiegiem obliczeń ani znanych z algorytmów genetycznych procesów selekcji. Dodatkowym utrudnieniem jest brak w tego typu systemach wiedzy globalnej oraz brak możliwości wykorzystania globalnej synchronizacji. Z tego też względu w systemach agentowych proces ewolucji jest sterowany bazując na wyczerpywalnych zasobach [53]. Proces wymiany zasobów pomiędzy środowiskiem a agentami wymaga zdefiniowania w systemie pewnej liczby zasobów, dzięki którym możliwa staje się ocena przydatności konkretnych agentów z punktu widzenia rozwiązywanego przez system problemu. Zasoby weryfikują zdolność agenta do

przetrwania w danym środowisku, gdyż schematy jego zachowań są określane poprzez rozkłady prawdopodobieństw, które są zależne od posiadanych przez niego zasobów. W najprostszym przypadku, zasoby są rozdzielane między agentów w taki sposób, iż te z wyższą wartością funkcji przystosowania do środowiska, dostają ich więcej, w związku z tym agenci mający niewystarczającą ilość zasobów, by uczestniczyć w procesach zachodzących w populacji (co jest określane przez warunki brzegowe, które określają minimalna ilość zasobów, konieczną do podjęcia działania przez agenta, mającego wpływ na działanie całego systemu), są usuwane z systemu a posiadane przez nie zasoby są przekazywane do środowiska bądź rozdzielane do innych agentów.

Podstawową odmianami wieloagentowych systemów ewolucyjnych są:

– **Ewolucyjny system wieloagentowy** (ang. *evolutionary multi-agent-system -EMAS*) [53] - w systemie tym każdy agent, jako podstawowa jednostka ewolucji, jest potencjalnym dostarczycielem rozwiązania. Podejście takie wymusza, od agenta zdolności do reprodukcji, w której cechy dziedziczone są generowane w sposób losowy (mutacje, rekombinacje). Dodatkowo system musi być wyposażony w mechanizm eliminacji agentów, najgorzej radzących sobie z postawionymi przed nimi zadaniami. W efekcie powstaje całkowicie zdecentralizowany system, który przy poprawnie zdefiniowanych mechanizmach selekcji, powinien doprowadzić do efektywnego rozwiązania postawionego przed nim problemu.

– **Stadny system wieloagentowy** (ang. *flock-based multi-agent system - FMAS*) [54] - w tym przypadku agent reprezentuje grupę osobników zwaną *stadem*. W ramach każdego stada realizowany jest proces ewolucji, taki jak np. klasyczny algorytm ewolucyjny, który został rozszerzony o możliwości migracyjne osobników, która odbywa się w ramach jednego izolowanego geograficznie regionu zwanego wyspą. W klasycznym algorytmie migracyjnym, w ramach takiej wyspy odbywa się proces ewolucyjny. W związku z tym selekcja najlepszych osobników ogranicza się do populacji wyspy, dlatego też wprowadzony zostaje operator migracji umożliwiający kopiowanie osobników między wyspami [68]. W przypadku FMAS, wprowadzona zostaje migracja osobników między stadami w obrębie jednej wyspy, oraz możliwa jest migracja całych stad pomiędzy wyspami [52].

Jedną z ważniejszych cech systemów agentowych jest łatwość tworzenia systemów hybrydowych, które łączą różnorodne techniki inteligencji obliczeniowej w obrębie jednego systemu [52]. I właśnie ta właściwość została wykorzystana w niniejszej pracy, gdzie wybrane fragmenty całego modelu w celach akceleracyjnych zostały zaimplementowane w układach GPGPU, co dokładnie zostało zaprezentowane w rozdziale 4.

2.2. Sieci neuronowe

Sieć neuronowa to model obliczeniowy o pewnych właściwościach upodobniających go do ludzkiego mózgu, w którym występują neurony, łączące je synapsy oraz układy nerwowe. Neurony będące podstawową komórką tworzącą układ nerwowy, komunikują się ze sobą poprzez przesyłanie impulsów

elektrochemicznych za pomocą synaps. W przypadku, gdy siła impulsu przekracza pewną wartość progową, powoduje to uwalnianie się związków chemicznych między synapsami. W reakcji na wspominane impulsy dochodzi do długoterminowych zmian długości połączeń, a mechanizm ten jest czynnikiem napędzającym proces uczenia się ludzkiego mózgu. W sieciach neuronowych największy wpływ na długoterminowe przechowywanie informacji mają wagi połączeń między neuronami, które rozmieszczone są w warstwach jak pokazano na rysunku 2.2. Pierwsza warstwa jest warstwą wejściową, odpowiadającą za wprowadzanie danych do sieci. Liczba neuronów w warstwie wejściowej jest zazwyczaj równa liczbie wejść do sieci. Za warstwą wejściową znajdują się warstwy ukryte. Zadaniem neuronów należących do poszczególnych warstw ukrytych jest realizowanie kolejnych etapów przetwarzania informacji wejściowych w informacje wyjściowe. W sieciach tego typu informacja zawsze przechodzi od warstw początkowych, poprzez ukryte do warstwy wyjściowej, stąd też sieci tego typu określane są mianem **sieci jednokierunkowych** (ang. *feed forward neutral network*). Wagi połączeń pomiędzy warstwami ukrytymi mają wpływ na rozpoznawanie przez sieć informacji zawartych w surowych danych treningowych. Ostatnią warstwę stanowi warstwa wyjściowa, w której generowana jest odpowiedź czyli prognoza modelu. W zależności od modelu wynik może być liczbą rzeczywistą w przypadku regresji, bądź prawdopodobieństwem w przypadku klasyfikacji. Rodzaj wyniku zależy od rodzaju funkcji aktywacji stosowanej w neuronach warstwy ukrytej. Funkcje aktywacji są wykorzystywane do generacji wartości wyjściowych przez neurony jednej warstwy, a następnie przesyłane do neuronów warstwy następnej. Argumentami i wyjściami tych funkcji są wartości skalarne. Wyjście funkcji jest aktywacją neuronu co stanowi analogię do wspomnianego wcześniej uwalniana związków chemicznych w ludzkim mózgu, w zależności od siły impulsu. Wybór funkcji aktywacji zależy od rodzaju problemu jaki został postawiony przed siecią. Dla sieci wielowarstwowych najczęściej stosowane są funkcje nieliniowe, gdyż neurony o takich charakterystykach wykazują największe zdolności do nauki, polegające na możliwości odwzorowania w sposób płynny dowolnej zależności pomiędzy wejściem a wyjściem sieci. Umożliwia to otrzymanie na wyjściu sieci informacji ciągłej, a nie tylko postaci binarnej. Funkcja aktywacji powinna zapewniać ciągłe przejście między swoją wartością minimalną a maksymalną, powinna być łatwa w obliczeniu i posiadać ciągłą pochodną oraz musi umożliwiać wprowadzenie parametru, dzięki któremu możliwe będzie regulowanie kształtu krzywej. Najczęściej stosowane funkcją aktywacji to: funkcja liniowa, funkcja progowa, sinusoida, cosinusoida, tangens hiperboliczny, sigmoida, funkcja ReLU. W warstwie wyjściowej najczęściej stosowaną funkcji aktywacji jest Softmax, który transformuje K-wymiarowy wektor wartości rzeczywistych do K-wymiarowego wektora z wartościami z zakresu (0;1), sumującymi się jedynki. Wyjście sofmtax reprezentuje prawdopodobieństwo przynależności wartości wejściowej do konkretnej klasy, dzięki czemu możliwe jest dokonanie klasyfikacji wieloklasowej. Połączenia warstw sieci neuronowej posiadają wagi, których modyfikacja jest podstawowym mechanizmem uczenia się sieci neuronowej. Wspomniane wagi są zbiorem liczb, który można określić mianem parametrów danej warstwy. Proces uczenia polega na znajdowaniu zbioru wartości wag wszystkich warstw sieci, tak by sieć poprawnie mapowała dane wejściowe na wyjściowe. Dostrajanie wartości wag, może odbywać się poprzez algorytm **propagacji wstecznej**, podczas której minimalizowany jest błąd, który określa różnicę między wartością wyjściową sieci, a wartością pożądaną. Błąd ten określany jest za pomocą **funkcji straty**, która jako parametry przyjmuje wartość pożądaną $\widehat{Y}$, oraz tę wyliczoną przez sieć Y. W przypadku regresji

najczęściej stosowana jest w funkcja w postaci logarytmicznej funkcji średnich kwadratów definiowanej jako:

$$\mathcal{L}(W,b) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}(log\widehat{y}_{ij} - logy_{ij})^2 \tag{2.1}$$

Natomiast w przypadku klasyfikacji najczęściej stosowany jest ujemny logarytm prawdopodobieństwa, który definiowany jest jako:

$$\mathcal{L}(W,b) = -\frac{1}{N}\sum_{i=1}^{N} y_i(log\widehat{y}_i + (1-y_i)log(1-\widehat{y}_i) \tag{2.2}$$

W algorytmie propagacji wstecznej, otrzymany błąd w warstwie ostatniej jest propagowany poprzez wszystkie warstwy ukryte, aż do warstwy wejściowej (czyli odwrotnie do przepływu informacji). Podczas propagowania błędu dokonuje się uaktualnienie wag każdej warstwy, w celu minimalizacji. Ponieważ wszystkie operacje sieci są różniczkowalne, a więc przy dostrajaniu wag możliwe jest wykorzystanie gradientowych metod optymalizacji, które są najczęściej wykorzystywane w procesie uczenia sieci neuronowych. W metodach tych aktualizacja wagi *w*, *j*-tego neuronu względem jego *i*-tego wejścia, odbywa się zgodnie z regułą najszybszego spadku, według formuły:

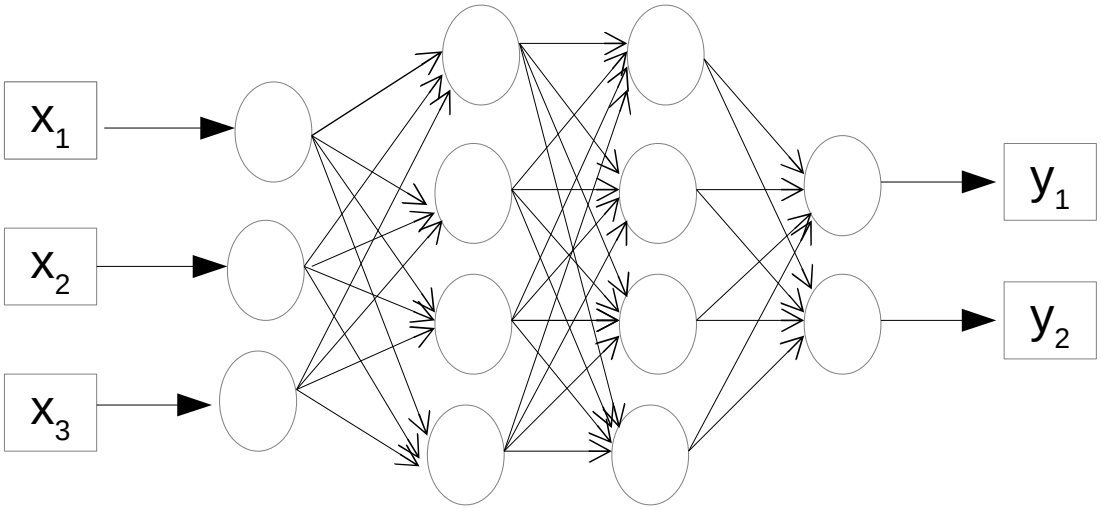
$$w_{ij} \longrightarrow w'_{ij} = w_{ij} - \eta\frac{\partial\mathcal{L}}{\partial w_{ij}} \tag{2.3}$$

Należy wspomnieć, iż prócz wartości wag neurony posiadają wyrazy wolne *b*, które również są aktualizowane w propagacji wstecznej zgodnie z formułą:

$$b_j \longrightarrow b'_j = b_{ij} - \eta\frac{\partial\mathcal{L}}{\partial b_j} \tag{2.4}$$

Regułę najszybszego spadku gradientu należy rozumieć jako podążanie krokami o rozmiarach η (współczynnik ten nazywany jest *learning rate* i winien być dobrze dobrany stosownie do architektury sieci), w kierunku antygradientu funkcji $\frac{\partial\mathcal{L}}{\partial w_{ij}}$ co oznacza podążanie w kierunku najszybszego spadku. W przypadku wstecznej propagacji, na początku obliczane są wagi dla warstwy ostatniej, po czym badany jest wpływ wag warstwy przedostatniej na błędne wagi warstwy ostatniej i na tej podstawie aktualizowane są wagi warstwy przedostatniej. Czynność ta jest powtarzana dla wszystkich warstw. Ogólnie rzecz ujmując, do aktualizacji wag warstwy *h-1*, konieczna jest znajomość wartości błędu warstwy *h*. W praktyce funkcja sieci neuronowych składa się z całego łańcucha operacji tensorowych (mianem tensor określa się dane przechowywane w wielowymiarowych tablicach), dla których należy określić wspólną pochodną. Zgodnie z zasadami matematyki pochodną takiego łańcucha oblicza się zgodnie z regułą łańcuchową definiowaną jako: f(g(x)) = f'(g(x))*g'(x). Zastosowanie reguły łańcuchowej podczas obliczania wartości gradientu sieci neuronowej prowadzi do uzyskania algorytmu propagacji wstecznej, który dzięki tej regule oblicza wpływ każdego parametru sieci na końcową wartość straty.

W celu wytrenowania sieci należy zadbać o parametry modelu i parametry regulacyjne, wpływające na jakość i szybkość działania sieci. Dobór hiper-parametrów ma na celu uzyskanie modelu, który nie jest nadmiernie ani niedostatecznie wytrenowany. Najważniejszymi rzeczami o jakie należy zadbać są liczba oraz rozmiar warstw ukrytych oraz czas trenowania sieci. Wszystko oczywiście zależy od zadania do jakiego sieć będzie wykorzystywana. Jeśli chodzi o liczbę warstw ukrytych, teoretyczną podstawę



Rysunek 2.2. Schemat sieci neuronowej

stanowią teorie zajmujące się aproksymacją funkcji wielu zmiennych. Traktując sieć jako układ aproksymujący dane uczące, zgodnie z twierdzeniem Kołmogorowa do aproksymacji dowolnej funkcji ciągłej wystarczy jedna warstwa ukryta, w przypadku odwzorowania nieliniowego należy użyć co najmniej dwóch takich warstw. Oczywiście przytoczone twierdzenie stawia tylko ograniczenia dotyczące minimalnej liczby warstw, których użycie powinno zagwarantować rozwiązanie problemu. Praktyczna zasada doboru liczby warstw ukrytych mówi, że im większy i im bardziej zróżnicowany zbiór treningowy, tym więcej warstw może zostać użytych bez obawy o nadmierne dopasowanie modelu. W przypadku doboru liczby neuronów warstw ukrytych sprawa nie jest tak oczywista jak w przypadku warstwy wyjściowej oraz wejściowej. Użycie za małej liczby pozbawi sieć środków niezbędnych do rozwiązania postawionego przed siecią problemu. Z kolei użycie zbyt wielu zwiększy czas uczenia i może przynieść efekt tzw. nadmiernego dopasowania, gdyż sieć będzie się uczyć nieistotnych cech zbioru uczącego, które są nieważne w populacji generalnej. Kolejną sprawą jest czas uczenia sieci, który powinien być tak długi aż wartość błędu uczenia oraz testowania przestanie maleć i zacznie utrzymywać się na stałym poziomie. Bardzo ważną rzeczą jest dostrajanie parametrów regulujących zwany **regularyzacją**. Jest to najbardziej czasochłonny etap, w którym dochodzi do modyfikacji modelu, trenowaniu go oraz weryfikacji skuteczności podjętych działań. Regularyzacja ma za zadanie zapobiegnięcie wystąpienia zjawiska *przetrenowania*, czyli doprowadzenie do sytuacji, gdy sieć uczy się zbyt dobrze pojedynczych obiektów, uzależniając zbytnio nauczone cechy od tych występujących w obiektach uczących, w tym także od cech drugorzędnych, nie dając podstaw do generalizacji. Do ważniejszych współczynników regularyzacyjnych należą:

– **momentum** - parametr umożliwiający pominięcie w algorytmie uczenia obszarów danych, w których poszukiwania najlepszego rozwiązania mogłyby utknąć w minim lokalnym. Dodatkowo zapobiega on zbyt chaotycznym zmianom wag, dzięki czemu algorytm uczy się bardziej konsekwentnie - kierunek zmian wag jest dłużej zachowany. Parametr ten jest wprowadzony do formuły

opisującej metodę najwyższego spadku (równanie 2.3) jako tzw. człon bezwładności: $\alpha \triangle w_{ij}^{t-1}$, który jest oznaczany jako *v* i stanowi o szybkości z jaką parametry poruszają się w swojej przestrzeni.

– **regularyzacja L1 i L2** - dzięki dodaniu współczynników L1=$\lambda|w|$, neurony korzystają tylko z najważniejszych wag i stają się mniej zależne od szumu. Natomiast współczynnik L2= λw^2 powoduje korzystanie z wszystkich wag w sposób bardziej równomierny - na wagi o zbyt dużej wartości zostaje nałożona kara, co sprzyja bardziej równomiernemu rozkładowi.

– **max-norm** - nałożenie twardych warunków ograniczających, które bezpośrednio wyznaczają nieprzekraczalny próg *d*, powyżej którego wagi są odpowiednio przeskalowywane. Wagi pozostające w zakresie nie są poddawane w tym przypadku procesowi korygowania.

– **dropout** - stanowi jedną z najprostszych i najbardziej skutecznych technik regularyzacji sieci neuronowych. Została zaproponowana w [95]. Działanie metody polega na losowym wybieraniu pewnej liczby cech wyjściowych warstwy podczas trenowania a następnie zastępowaniu ich zerami. Ułamek określający część wyzerowanych cech nazywany jest współczynnikiem porzucania (ang. *dropout rate*). Motywacją takiego postępowania jest zablokowania budowania bardzo szczegółowych zależności między pobliskimi neuronami. Technika ta stanowi ważną część rozdziału 6 i zostanie tam bardziej szczegółowo opisana.

Bardzo ważnym parametrem procesu uczenia sieci neuronowej jest **szybkość uczenia**, który ma wpływ zarówno na stabilność jak i skuteczność tego procesu. Pożądane jest by parametr ten miał dużą wartość na początku treningu, która wraz z upływem czasu, gdy proces zacznie się stabilizować, powinna spadać. Wpływ na szybkość uczenia ma zastosowanie wcześniej wspomnianego współczynnika *momentum*. Spadek szybkości uczenia nie powinien być zbyt gwałtowny, gdyż mogłoby to doprowadzić do zbyt wczesnego zakończenia treningu.
Istotnym aspektem procesu uczenia jest zadbanie o prawidłową inicjalizację wag, gdyż jest to najważniejszy punkt startowy w procesie uczenia. Oczywiście wszystkie neurony tej samej warstwy otrzymują ten sam wektor wejściowy, z tego też względu muszą one zostać zainicjalizowane różnymi wartościami wag, gdyż w przeciwnym przypadku na wyjściu wszystkie wartości byłyby takie same. Błędna inicjalizacja wag może doprowadzić do znacznego wydłużenia procesu uczenia, braku jego zbieżności oraz spowodować, że błąd generalizacji może się istotnie zmieniać dla tej samej architektury i stałych hiperparametrów w zależności od konkretnej realizacji inicjalizacji losowej na co została zwrócona uwaga w [36]. Sposobem na przeciwdziałanie nieporządnym efektom jest wprowadzanie tzw. *łamanej symetrii*, polegającej na losowaniu początkowych wartości wag z zakresu, który musi być wystarczająco duży żeby złamać symetrię, a jednocześnie na tyle mały by nie wprowadzić sieć w stan, z którego ciężko będzie ją oduczyć. Duże wartości wag wprowadzają silniejszy efekt „łamania symetrii" i powodują brak efektu zanikania sygnału uczącego, ale jednocześnie zwiększają ryzyko nadmiernego wzrostu wartości gradientu (tzw. exploding gradient), wprowadzają zbytnią czułość wyjściowych predykcji na małe zmiany parametrów wejściowych, oraz mogą wydłużyć czas dostosowywania wag jeżeli ich docelowy rozkład w punkcie zbieżności uczenia jest znacząco inny od tego założonego przy inicjalizacji. W związku z tym w litera-

turze [31] [42] [62] zostały zaproponowane techniki uzależniające inicjalizację wag od liczby neuronów poszczególnej warstwy. Oznaczając jako f_{in} liczbę połączeń wchodzących, f_{out} liczbę połączeń wychodzących, warianty rozkładu prawdopodobieństwa, zaproponowane we wspomnianej literaturze mają postacie:

– LeCun normal[62] :

$$w \sim \mathcal{N}(0, \frac{1}{f_i n}) \tag{2.5}$$

– He normal[42] :

$$w \sim \mathcal{N}(0, \frac{2}{f_i n}) \tag{2.6}$$

– Xavier normal[31] :

$$w \sim \mathcal{N}(0, \frac{2}{f_i n} + f_{out}) \tag{2.7}$$

Przedstawione techniki są powiązane z budową sieci, więc ten czynnik powinien być głównie brany pod uwagę podczas wyboru sposobu inicjalizacji wag sieci neuronowej.
Kolejnym ważną rzeczą o jaką należy zadbać podczas trenowania sieci jest dobór metody optymalizacyjnej, której głównym celem jest poszukiwanie parametrów θ, które określają model, a których optymalizacja ma na celu zmniejszanie łącznej wartości funkcji kosztu modelu określanej wzorem (M-l.próbek uczących, Y-macierz etykiet zbioru uczącego):

$$J(\theta, X, Y) = -\frac{1}{M} \sum_{i=1}^{M} \mathcal{L}^{[i]}. \tag{2.8}$$

Jak zostało wyżej wspomniane najczęściej stosowanymi metodami służącymi do minimalizacji funkcji kosztu są metody gradientowe. Najprostszą ich postacią jest metoda najszybszego spadku (ang. *batch gradient descent*), która dokonuje wyznaczenia gradientu na bazie całego zbioru uczącego, przez co mimo bardzo dużej dokładności jest metodą bardzo powolną, przez co kompletnie niepraktyczną. Z tego względu zostały wprowadzone inne metody, opierające swoje działanie na mniejszych za to częstszych krokach w kierunku gradientu przybliżonego. Pierwszą tego typu metodą jest *Stochastic gradient descent*, która estymuje wartość gradientu używając tylko jednej próbki uczącej. Powoduje to iż aktualizacja wag staje się zarazem bardzo szybka i bardzo niedokładna. Z tego względu lepszym rozwiązaniem wydaje się stosowanie metody estymującej gradient na podstawie *m* próbek uczących, noszącej nazwę *mini-batch gradient descent*. Zastosowanie pewnej liczby próbek uczących jest dużo szybsze niż zastosowanie całego zbioru, a zarazem dużo bardziej stabilne niż w przypadku stosowania pojedynczej próbki. Dobór parametru *m* jest kwestią niejednoznaczną i często odbywa się metodą prób i błędów. Duża wartość tego parametru prócz zwiększania wymaganej liczby obliczeń, powoduje zwiększenie zapotrzebowania pamięciowego co jest bardzo ważne w przypadku akceleratorów GPGPU. Stochastyczny spadek gradientu $\hat{g}_t$ w pojedynczej iteracji dla *m* próbek uczących wyraża się wzorem:

$$\hat{g}_{(t)} = \frac{1}{m} \triangle_{\theta_{(t)}} \sum_{i}^{m} \mathcal{L}(f(x^{[i]}; \theta_{(t)}), y^{[i]}) \tag{2.9}$$

Natomiast aktualizacja odbywa przy użyciu wyrażenia:

$$\theta_{(t+1)} = \theta_{(t)} - \eta \hat{g}_{(t)} \tag{2.10}$$

Bardzo ważną rolę w procesie optymalizacyjnym odgrywa opisany wyżej parametr *momentum*, który w kolejnej optymalizacji zwanej *Nesterov momentum* [73], uwzględnia dodatkowo wpływ wartości *momenutm* na zmianę gradientu wyliczanego w danym kroku. Poniższe wzory określają estymatę gradientu kolejno dla **momentun** oraz *Nesterov momentum*

$$v_{(t)} \longleftarrow \alpha v_{(t-1)} - \eta \triangle_{\theta_{(t)}} \frac{1}{m} \sum_{i}^{m} \mathcal{L}(f(x^{[i]}; \theta_{(t)}), y^{[i]}) \tag{2.11}$$

$$v_{(t)} \longleftarrow \alpha v_{(t-1)} - \eta \triangle_{\theta_{(t)}} \frac{1}{m} \sum_{i}^{m} \mathcal{L}(f(x^{[i]}; \theta_{(t)} + \alpha v), y^{[i]}) \tag{2.12}$$

W obu przypadkach aktualizacja odbywa się za pomocą zależności:

$$\theta_{(t+1)} = \theta_{(t)} + v_(t) \tag{2.13}$$

Kolejna metoda optymalizacyjna została zaproponowana w [20] i nosi nazwę *AdaGrad*. Algorytm ten polega na szybkim wygaszaniu tempa uczenia, poprzez skalowanie parametrów odwrotnie proporcjonalnie do pierwiastka z sumy poprzednich kwadratów gradientów. Zabieg ten pozwala uzyskać większą optymalizację w kierunku przestrzeni parametrów o mniejszym nachyleniu, gdyż mają one mały spadek szybkości uczenia uczenia się. Jeśli *r* oznacza skumplowane wartości kwadratów gradientu, to wówczas obliczanie nowej wartości odbywa się za pomocą równań:

$$r_{(t)} \longleftarrow r_{(t-1)} + \hat{g}_{(t)} \odot \hat{g}_{(t)} \tag{2.14}$$

$$\theta_{(t+1)} \longleftarrow \theta_{(t)} - \frac{\eta}{\sqrt{r_t + \epsilon}} \odot \hat{g}_{(t)} \tag{2.15}$$

gdzie ϵ oznacza stałą o małej wartości, wprowadzoną w celu stabilności numerycznej. Operacje pierwiastkowania, dzielenia i wymnażania są wykonywane z osobna dla każdego elementu wektora. Zastosowanie tego podejścia od samego początku uczenia, poprzez zbyt szybki wzrost mianownika, może spowodować zanik uczenia. Z tego względu wprowadzono nowy algorytm o nazwie *RMSProp*[116], który zamiast sumy kwadratów proponuje użycie średniej zanikającej wykładniczo, aby odrzucić wartości te najbardziej historyczne. W algorytmie tym zostaje wprowadzony parametr kontrolujący skalę długości ruchomej średniej ρ. Stąd wyrażenie obliczające *r* przyjmuje postać:

$$r_{(t)} \longleftarrow \rho r_{(t-1)} + (1-\rho)\hat{g}_{(t)} \odot \hat{g}_{(t)} \tag{2.16}$$

$$\theta_{(t+1)} \longleftarrow \theta_{(t)} - \frac{\eta}{\sqrt{r_t + \epsilon}} \odot \hat{g}_{(t)} \tag{2.17}$$

Kolejnym adaptacyjnym algorytmem optymalizacyjnym, który może być traktowany jako kombinacja wcześniej opisanych *RMSProp* oraz *momentum*, jest algorytm popularnie znany jako *Adam* [51]. Algorytm ten dokłada estymaty momentu pierwszego rzędu z wagami w postaci wykładniczej. W związku z tym Adam obejmuje korekty estymacji momentu zarówno pierwszego i drugiego rzędu (RMSProp zawiera tylko korekty drugiego rzędu). Aktualizacja wag odbywa się za pomocą równań:

$$s_t \longleftarrow \rho_1 s_{(t-1)} + (1-\rho_1)\hat{g}_{(t)} \tag{2.18}$$

$$r_{(t)} \longleftarrow \rho_2 r_{(t-1)} + (1-\rho_2)\hat{g}_{(t)} \odot \hat{g}_{(t)} \tag{2.19}$$

Autorzy rozwiązania wprowadzają korektę do obciążenia momentu, co odgrywa szczególną rolę na początku szkolenia, gdzie może być zbyt duże odchylenie w stronę zera. Korekta ta w kroku *t* odbywa się poprzez:

$$\hat{s}_{(t)} \longleftarrow \frac{s_{(t)}}{1-\rho_1^t} \tag{2.20}$$

$$\hat{r}_{(t)} \longleftarrow \frac{r_{(t)}}{1-\rho_2^t} \tag{2.21}$$

Ostatecznie aktualizacja parametrów wyraża się wzorem:

$$\theta_{(t+1)} \longleftarrow \theta_{(t)} - \frac{\eta \hat{s_{(t)}}}{\sqrt{\hat{r}_t + \epsilon}} \tag{2.22}$$

Omówione wyżej algorytmy optymalizacyjne są obecnie najczęściej używane, nie istnieje jednak jednoznaczna odpowiedź, który z nich jest najlepszy. W związku z tym wybór, którego należy użyć, zależy od modelu sieci jaki ma zostać użyty. Warto wspomnieć, że omówione algorytmy opierają się na estymowaniu pochodnych pierwszego rzędu. Prócz tego istnieją jeszcze metody estymujące pochodne drugiego rzędu, jednak ze względu na ich złożoność pamięciową ograniczona jest możliwość ich powszechnego stosowania w sieciach zawierających parametry liczone w mega-bajtach. Z tego względu metody te zostają pominięte w niniejszej pracy.

2.2.1. Konwolucyjne sieci neuronowe

Podstawą konwolucyjnych sieci neuronowych (ang. *convolutional neutral networks - cnn*) jest operacja konwolucji, inaczej nazywana splotem, która w analizie matematycznej jest działaniem na dwóch funkcjach *f* oraz *g*, którego wynikiem jest inna funkcja *s*, która może być postrzegana jako zmodyfikowana wersja jednej z oryginalnych funkcji, co można zapisać za pomocą równania:

$$s(t) = (f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t-\tau) d\tau \tag{2.23}$$

Powyższe równanie dla dyskretnych wartości *t* jest definiowane jako:

$$s(t) = (f * g)(t) = \sum_{\tau=-\infty}^{\infty} f(\tau) g(t-\tau) \tag{2.24}$$

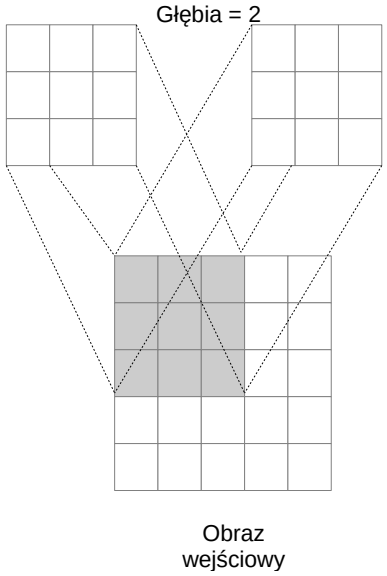
W konwolucyjnych sieciach neuronowych, operacja konowulucji odnosi się do warstwy konwolucyjnej, która jest zdefiniowana poprzez zbiór *K* filtrów o rozmiarach $R \times S$ oraz głębokości *C*. Funkcja *f* stanowi wejście do modelu, którego liczba wymiarów determinuje rodzaj konwolucji, który może być 1D - wykorzystywany do przetwarzania głosu oraz tekstu, 2D - służący do przetwarzania obrazów, bądź 3D - szczególnie przydatny do przetwarzania obrazów pochodzących z rezonansu magnetycznego (ang. *magnetic resonance imaging - MRI*). Dane wejściowe reprezentowane są poprzez *N*-elementowy *batch*, zawierający *C* wymiarowe dane(tekst, audio obrazy etc.) o rozmiarach $H \times W$. Funkcja *g* pełni rolę kernela (nazywanego też filtrem), posiadającego taką samą liczbę wymiarów jak dane wejściowe, zawierającego zdolne do nauki wagi. Wynik konwolucji nosi miano mapy atrybutów (ang. *feature map)* lub mapy aktywacji *(activation map)*. W praktyce w uczeniu maszynowym wykorzystuje się dyskretną formę konowolucji. Wejście oraz zbiór wag są zapisane w postaci 4-wymiarowego tensora. Oznaczając kernel

jako *W*, a wejście jako *I* konwolucja pojedynczej warstwy jest obliczana poprzez:

$$Out_{n,i,j,k} = \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} W_{k,c,r,s} I_{n,c,i+r,j+s} \tag{2.25}$$

Wynik konwolucji jest sumowany z wyrazem wolnym *b*, a następnie aplikowana jest na nim funkcja aktywacji, najczęściej w postaci *ReLU* bądź *sigmoid*. Główną różnicą między konwolucyjnymi sieciami neuronowymi, a a sieciami w pełni połączonymi (z warstwami gęstymi), jest sposób połączenia warstw. W standardowym przypadku w warstwach ukrytych, każdy neuron połączony jest z każdym neuronem warstwy poprzedzającej (rysunek 2.2). Podejście to jest przydatne w sytuacji, gdy warstwa wejściowa jest zbiorem wielu różnorakich atrybutów charakteryzujących zbiorczo daną obserwację ponieważ warstwy gęste uczą się cech parametrów globalnych. Jest to problematyczne ze względu na ogromną liczbę koniecznych połączeń, jakie muszą wystąpić w takiej sieci. Przykładowo dla dwu-wymiarowego obrazka, o wymiarach 100x100 tylko w pierwszej warstwie musi znaleźć się łącznie 10 mln połączeń, co oczywiście stanowi spory problem. W przypadku cnn problem ten jest rozwiązany poprzez zastosowanie lokalnych połączeń neuronów. Wartość aktywacji poszczególnych neuronów w warstwie ukrytej zależy wyłącznie od paru neuronów z warstwy poprzedniej, znajdujących się w określonym lokalnym otoczeniu danego neuronu. Otoczenie to jest nazywane *polem postrzegania neuronu (ang. receptive field)*. Dzięki takiemu podejściu splotowe warstwy ukryte, uczą się tylko lokalnych wzorców, co zostało zainspirowane biologią, a dokładniej obszarem mózgu odpowiedzialnym za widzenie. Komórki w tym obszarze są wrażliwe na małe podregiony z odbieranych sygnałów wejściowych poprzez pole widzenia i aktywują się, gdy w danym subregionie nastąpi zdarzenie np. zostaną wykryte jakieś wzorce. Zastosowanie lokalnych połączeń znacznie redukuje liczbę parametrów koniecznych do reprezentacji sieci. Mniejsza liczba parametrów przyśpiesza proces uczenia oraz stanowi swego rodzaju formę regularyzacji zabezpieczającej przed nadmiernym dopasowaniem. Operowanie przez neurony splotowe tylko na niewielkich wycinkach poprzedniej warstwy powoduje zastosowanie tych samych wartości wag dla różnych danych, co w znaczący sposób ułatwia wykrywanie wzorców. Działanie filtrów konwolucyjnych może być postrzegane jako przesuwanie pojedynczego filtru zawierającego wagi charakteryzujące konkretną cechę, względem zmieniającego się ciągu danych wejściowych. Filtr taki działa jak detektor charakteryzowanej przez niego cechy, która zostanie wykryta niezależnie od jej położenia w całym obszarze danych wejściowych. Kaskadowe ułożenie warstw konwolucyjnych pozwala na detekcję cech wyższego rzędu - kolejne warstwy sieci tworzą reprezentacje danych o coraz wyższym poziomie abstrakcji cech. Ponadto podczas dodawania kolejnych warstw reprezentacja zaczyna reagować na duże obszary pikseli. W wyniku połączenia ze sobą odpowiedniej liczby warstw możemy dojść do globalnej reprezentacji całego obrazu wejściowego. Jak zostało wcześniej powiedziane, każda warstwa konwolucyjna zawiera zestaw definiujących ją parametrów. Pierwszym parametrem jest wielkość filtru czyli jego wysokość i szerokość, oznaczone wcześniej jako $R \times S$. Dobór wielkości filtru ma bardzo duży wpływ na dokładność klasyfikacji. Duże filtry generują dane wyjściowe o większych rozmiarach, a skuteczne zarządzanie rozmiarem danych wyjściowych poszczególnych warstw konwolucyjnych prowadzi do zwiększania wydajności sieci. Rozmiary filtrów muszą być mniejsze od rozmiarów danych wejściowych. Najczęściej spotykane filtry, jeśli chodzi o przetwarzanie obrazów, to 3x3 bądź 5x5. Kolejnym parametrem jest *głę-*

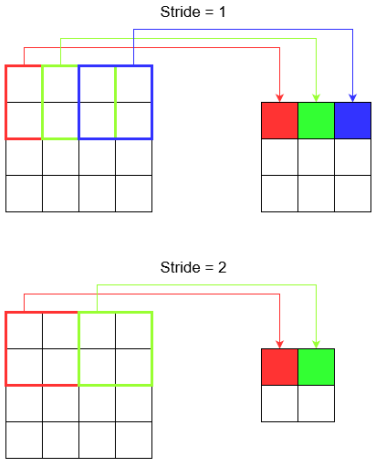
bia (*depth*) oznaczana jako *C*, który wpływa na liczbę neuronów w warstwie konwolucyjnej, połączonych z tym samym obszarem wejścia. Głębia nie opisuje zawartości danych, ale określa liczbę kanałów niezbędnych do jego dokładnego opisu. W przypadku obrazów parametr ten wynosi 3, gdyż definiuje on kanały kolorów - niebieski, czerwony i zielony (RGB), dlatego też pierwsza warstwa konwolucyjych sieci neuronowych przeznaczonych do przetwarzania obrazów, ma ustawiony ten parametr właśnie na 3. W celu dokładnego przeanalizowania obrazu sieć musi rozpoznawać cechy, które są trudne do opisania, a jest to możliwe poprzez zwiększanie wartości głębi w późniejszych warstwach, ponieważ pozwala to zakodować więcej informacji opisujących dane wejściowe. Przyjęcie zbyt niskiej wartości parametru głębi prowadzi do słabej efektywności sieci, ponieważ sieć nie będzie dysponowała odpowiednią liczbą kanałów do dokładnego scharakteryzowania danych wejściowych. Zbyt duża wartość natomiast prowadzi do konieczności przeprowadzenia bardzo dużej liczby operacji matematycznych, co osłabia wydajność sieci. Działanie tego parametru obrazuje rysunek 2.3. Parametr określający odstęp pomiędzy neuronami



Rysunek 2.3. Graficzna interpretacja parametru *Depth*

nosi nazwę *Stride*. Można go rozumieć jako wartość przesunięcia filtru po danych wejściowych, przy każdorazowym zastosowaniu funkcji konwolucji. Mała wartość tego parametru prowadzi do większego nakładania się na siebie pól, co prowadzi do utworzenia większych pakietów wyjściowych. Zwiększanie jego wartości prowadzi do mniejszego nakładania się i redukcji rozmiarów danych wyjściowych. Strojenie tego parametru jest poszukiwaniem kompromisu pomiędzy dokładnością, a rozmiarem danych wyjściowych. Graficzna interpretacja tego parametru znajduje się na rysunku 2.4. Ostatnim parametrem jest tzw. *padding*. Odpowiada on za uzupełnianie zerami skrajnych wartości każdego pola odbiorczego. Umożliwia to zrównanie rozmiarów danych danych wejściowych z wyjściowymi, co ułatwia zarządzanie parametrami *depth* i *stride*. Gdyby nie ten zabieg, spolt miałby tendencję do stopniowego zniekształcenia treści znajdujących się na krawędziach filtra.

Definiując sieć konwolucyjną należy dobrać odpowiednie wartości wyżej opisanych parametrów. Aby



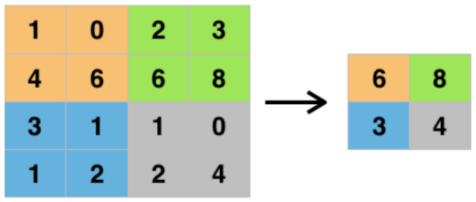
Rysunek 2.4. Graficzna interpretacja parametru *stride*

możliwe było dobranie wartości parametrów dla warstwy *O*, należy znać rozmiar danych wyjściowych generowanych przez warstwę *O-1*, a przestrzenny rozmiar danych wyjściowych, generowanych przez tę warstwę będzie miał rozmiar:

$$O = \frac{rozmiar\_wejcia - rozmiar\_filtru + 2 * padding}{stride} + 1 \tag{2.26}$$

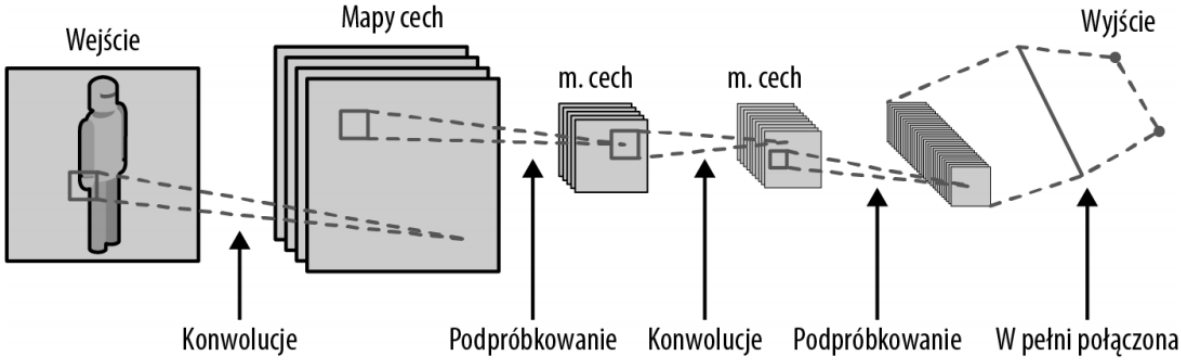
Prócz warstw konwolucyjnych, w modelu występują jeszcze warstwy *pooling*, których zadaniem jest agregowanie wielu wartości mapy cech do pojedynczej wartości najczęściej za pomocą funkcji maksimum (max-pooling), średniej (mean-pooling) lub sumy (sum-pooling). Dzięki temu zabiegowi zostaje zredukowany rozmiar danych wejściowych, poprzez wybór jednej z $n \times n$ wartości. Najczęściej spotyka się filtr o rozmiarach 2×2 z krokiem 2. Używając takiego filtru dla obrazku wejściowego o rozmiarach 32×32, na wyjściu znajdzie się obraz o rozmiarach 16×16. Inną zaletą omawianej warstwy jest odporność na występowanie wielu małych odchyleń. *Pooling* zapobiega przeuczeniu oraz sprawia, że sieć przestaje przywiązywać wagę do dokładnego położenia cech i potrafi je bardziej uogólnić. Warstwa ta nie ma parametrów. Działanie warstwy *pooling* z funkcją max, przy użyciu filtru 2×2 z krokiem 2, obrazuje rysunek 2.5.

Warstwy konwolucyjne oraz pooling stanowią trzon większości wykorzystywanych w praktyce modeli



Rysunek 2.5. Warstwa pooling z funkcją max

konwolucyjnych. Na wyjściu sieci zazwyczaj znajduje się kilka warstw w pełni połączonych (ang. *fully connected - fc*). Tego typu warstwy są wykorzystywane do wyliczenia ocen klas stanowiących wynik działania sieci. W tych warstwach wszystkie neurony są połączone wzajemnie ze sobą, jak również z



Rysunek 2.6. Struktura konwolucyjnej sieci neuronowej [74]

neuronami warstwy poprzedniej. Na wyjściu sieci użyta jest zazwyczaj wcześniej opisana funkcja *Softmax*, a rozmiar wyjścia wynosi $[1\times1 \times N]$, gdzie N oznacza liczbę klas wyjściowych. W rozdziale 6 zostaną dokładniej opisane niektóre architektury CNN. Na rysunku 2.6 został przedstawiony ogólny schemat sieci konwolucyjnej, składający się z opisanych w tym rozdziale warstw. Warto nadmienić, że prócz operacji konwolucji istnieje operacja nazywana konwolucją transponowaną (ang. *transposed convolution*) lub konwolucją ze skokiem ułamkowym (ang. *fractional stride convolution*) [93]. Głównym celem tej operacji jest przejście z reprezentacji o mniejszym rozmiarze z powrotem do reprezentacji większej. Celem procesu uczenia takiej warstwy jest takie ustalenie wag, aby możliwe było dokonanie przekształcenia odwrotnego do splotu, jednak wagi obu tych operacji nie muszą być ze sobą w żaden sposób powiązane. Rozmiar wyjścia takiej warstwy definiowany jest za pomocą wyrażania:

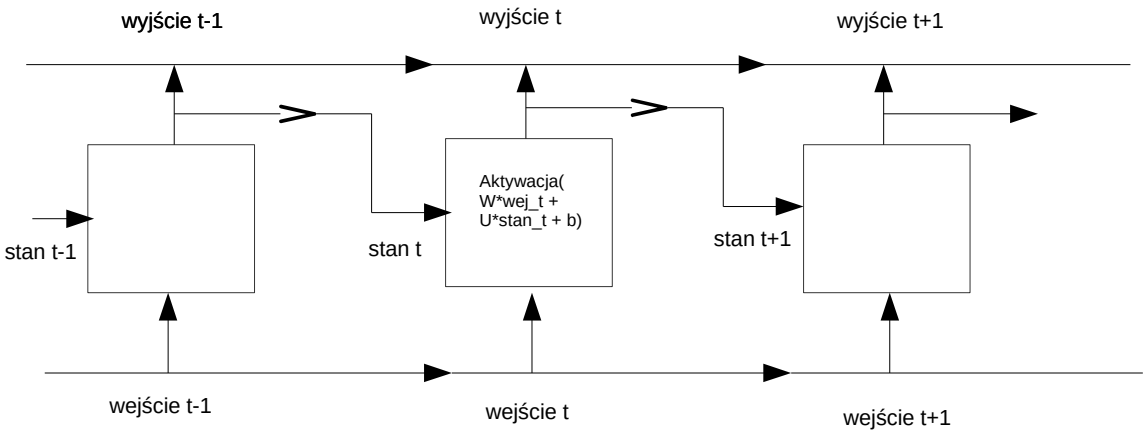
$$O = stride(rozmiar\_wejcia - 1) + rozmiar\_filtru - 2 * padding \tag{2.27}$$

Warstwy tego typu znalazły zastosowanie między innymi w auto-dekoderach splotowych(ang. *convolutional autoencoders*) [86] w sieciach typu GAN (ang. *generative adversarial networks*) [88].
Ostatnią rzeczą, o której warto wspomnieć w kontekście sieci konwolucyjnych jest operacja splotu wykorzystująca filtr o rozmiarach 1×1. Jak zostało to wyżej wspomniane, siłą operacji splotowych jest odporne na przesunięcia reagowanie na określone wzorce. Uzyskuje się to stosując filtry o rozmiarach łączących, w przypadku obrazów, kilka pikseli. Filtr o rozmiarze 1×1 traci tę właściwość, gdyż piksele przetwarzane są niezależnie od siebie i każdemu pikselowi przyporządkowywany jest dokładnie jeden piksel wyjściowy. Główną korzyścią ze stosowania tego typu filtrów jest redukcja wymiarowości danych generowanych przez warstwy zawierające dużą liczbę filtrów splotowych. Zastosowany do takich warstw zmniejsza głębokość danych nie zmieniając ich rozmiaru. Użycie tego specjalnego rodzaju konwolucji zostało przedstawione w rozdziale 6, gdzie została zbadana możliwość jego akceleracji względem cuDnn w układach GPGPU.

2.2.2. Rekurencyjne sieci neuronowe

Opisane dotychczas sieci jednokierunkowe oraz splotowe, nie posiadały zdolności zapamiętywania stanu, to znaczy, że dane wejściowe są w nich przetwarzane niezależnie od siebie oraz nie są zachowywane żadne zaobserwowane relacje między nimi. Rekurencyjne sieci neuronowe (ang. *Recurrent neural network - RNN*), podczas przetwarzania kolejnych elementów, utrzymują wewnętrzny stan czyli informacje o dotychczasowych obserwacjach tego co zostało dotychczas przetworzone. Zdolność zapamiętywania w sieciach RNN jest realizowana poprzez pętlę w połączeniach (zwane również połączeniami rekurencyjnymi), obejmująca sąsiednie kroki czasowe. Dzięki połączeniom rekurencyjnym możliwe jest modelowanie działania czasowego w danych, które ze swojej natury są zależne czasowo czyli szeregów czasowych takich jak tekst czy audio. Dane w tych dziedzinach są uporządkowane oraz wrażliwe kontekstowo, gdyż wartość na danej pozycji jest uzależniona od wartości ją poprzedzających, jak również po niej następujących. Połączenia zastosowane w tego typu sieciach pozwalają na identyfikowanie występujących efektów czasowych. Sieci RNN posiadają dodatkowy parametr *krok czasowy*, służący do identyfikowania zależności czasowych między danymi, poprzez połączenia pomiędzy krokami czasowymi. Podczas przepływu danych przez sieć, w każdym kroku czasowym dzięki połączeniom rekurencyjnym, kolejne węzły odbierają aktywację z bieżącego wektora wejściowego, jak również z ukrytych węzłów w stanie poprzednim. W związku z czym wektor z poprzedniego kroku czasowego może wpływać poprzez połączenia rekurencyjne na wynik bieżącego kroku, co zostało zobrazowane na rysunku 2.7.

W sieciach typu RNN w każdym kroku czasowym zmienia się liczba wektorów wejściowych, a każdy



Rysunek 2.7. Struktura rekurencyjnej sieci neuronowej

wektor może składać się z różnej liczby kolumn. W zależności od operacji przeprowadzanych na danych można wyróżnić następujące rodzaje RNN:

– **jeden do wielu** - dane wejściowe w przeciwieństwie do wyjściowych nie są sekwencją. Przykładem jest opisywanie obrazów, gdzie wejście to obraz, a na wyjściu znajduje się jego opis,

– **wiele do jednego** - dane wejściowe to sekwencja, a wyjście to wektor o stałym rozmiarze. Dla przykładu na wejście podawany jest fragment tekstu, a na wyjściu znajduje się słowo kluczowe,

opisujące fragment tekstu. Innym przykładem jest sekwencja nut podana na wejściu, natomiast model zwraca gatunek muzyki do jakiego zalicza się dany utwór,

– **wiele do wielu** - dane wejściowe i wyjściowe składają się z sekwencji. Przykład to tłumaczenie jednego języka na inny, generowanie nowego utworu muzycznego na podstawie innego itp.

Problemem z opisaną wyżej architekturą sieci rekurencyjnych jest ich zbyt prosta struktura, przez co sieć ma problemy z uczeniem się bardziej rozległych czasowo zależności, które konieczne są gdy rozwiązywany problem wymaga szerokiego kontekstu. Jest to wynikiem zanikającego gradientu [76], który powstaje w wyniku wprowadzenia zbyt dużej ilości warstw by zachować długotrwałe zależności czasowe, co prowadzi do gromadzenia się zbyt dużej ilości gradientów, a wówczas nie możliwe staje się modelowanie długookresowych zależności. Rozwiązaniem tego problemu stało się wprowadzenie opisanych poniżej warstw typu **LSTM**.

2.2.2.1. Long Short Term Memory (LSTM)

LSTM to specjalny rodzaj sieci RNN, zdolny do przechowywania długoterminowych zależności, pomiędzy danymi wejściowymi, a wyjściowymi wynikami zwracanymi przez sieć. Najważniejszym komponentem jest komórka pamięci, reprezentowana przez ukrytą warstwę w rekurencyjnej sieci neuronowej. W skład komórki wchodzą specjalne *bramki*, dzięki którym możliwe jest zapisywanie, odczytywanie oraz usuwanie informacji. Decyzja o podjętej względem informacji akcji jest podejmowana na podstawie przypisanej do informacji wagi, która dobierana jest w procesie uczenia, gdzie wraz z upływem czasu warstwa LSTM uczy się, które informacje są bardziej bądź mniej ważne. Schemat pojedynczej komórki LSTM dobrze ilustruje rysunek 2.8. Każda jednostka LSTM posiada następujące rodzaje wejść:

– połączania przyjmujące pamięć z poprzedniego kroku czasowego $c^{<t-1>}$

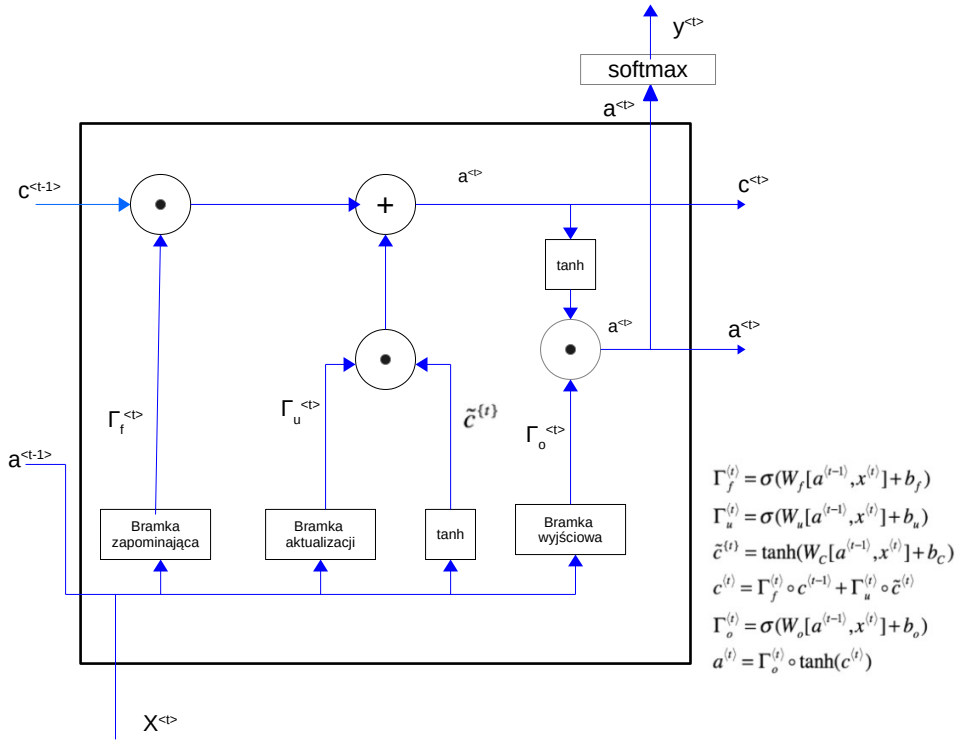
– połączenia z wyjściem poprzedniej warstwy $a^{<t-1>}$

– wektor x^t w kroku czasowym t

Komórka LSTM posiada trzy różne typy bramek:

– **bramka zapominająca**(ang. *forget gate*) - pozwala komórce pamięci na zapominanie jej zawartości. Do bramki zapominającej trafiają wektory będące rezultatem przemnożenia funkcji aktywacji (zwracającej wyniki z przedziału (0,1)), której wejściem były wektory wejściowe przemnożone przez wagi oraz wcześniejszego stanu pamięci. Wartości ze stanu poprzedniego zostaną zapamiętane w momencie, gdy funkcja aktywacji zwróci wartości bliskie jeden, w przeciwnym przypadku, gdy wyjścia z tej funkcji są bliskie zeru wówczas wartości stanu poprzedniego zostają wymazywane.

– **bramka aktualizacji** (ang. *update gate*) - jej zadaniem jest aktualizacja stanu komórki $c^{<t>}$. Funkcją aktywacji jest tangens hiperboliczny (przyjmuje wartości z przedziału (-1;1)), za pomocą którego obliczane są wartości, które mogą być wstawiane do stanu komórki.

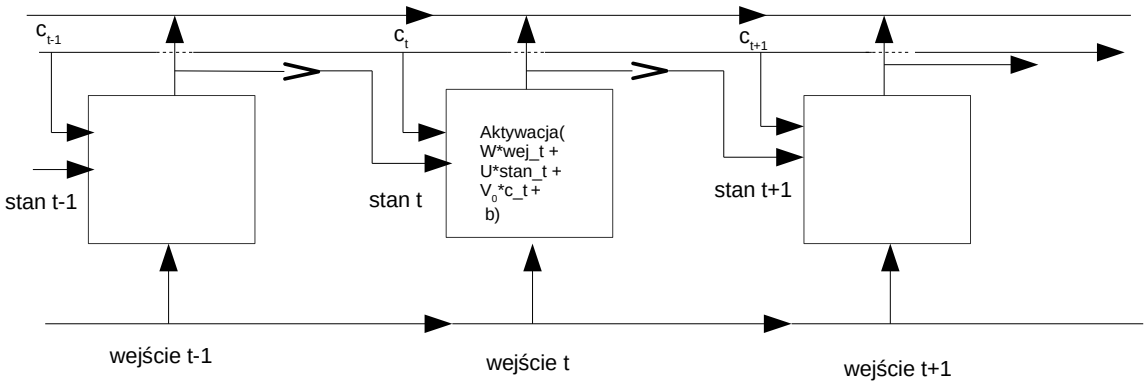


Rysunek 2.8. Schemat komórki sieci LSTM

– **bramka wyjściowa** (ang. *output gate*) - ostatni element komórki LSTM, składający się z dwóch komponentów: funkcji aktywacji w postaci *tanh* oraz wyjściowej funkcji sigmoidalnej.

Na rysunku 2.8 został pokazany przepływ informacji przez komórkę LSTM wraz z równaniami dzięki którym obliczane są wyjścia poszczególnych modułów komórki. Poruszając się zgodnie ze strzałkami, dane przechodzą przez bramę zapomnienia, gdzie decyduje się które informacje mają zostać zachowane, na podstawie przemnożenia wektora wejściowego poprzez wektor wyjściowy z poprzedniej komórki, po czym po użyciu funkcji aktywacji wartości bliskie jeden tworzą tzw. *wektor wartości kandydujących* i zostają zapamiętane. W kolejnym kroku dokonuje się aktualizacja stanu komórki, określonego przez $c^{<t-1>}$, poprzez wektor $c^{<t>}$, który jest odbierany przez koleją komórkę LSTM w kolejnym kroku czasowym. Ostatnim krokiem jest zwrócenie wartości $y^{<t>}$, która jest wartością przewidywaną przez komórkę LSTM w czasie t. Wartość ta stanowi jednocześnie wejście do kolejnej warstwy ukrytej, jak również do kolejnej komórki LSTM. Schemat przepływu danych poprzez sieć zawierającą komórki LSTM, obrazuje rysunek 2.9.

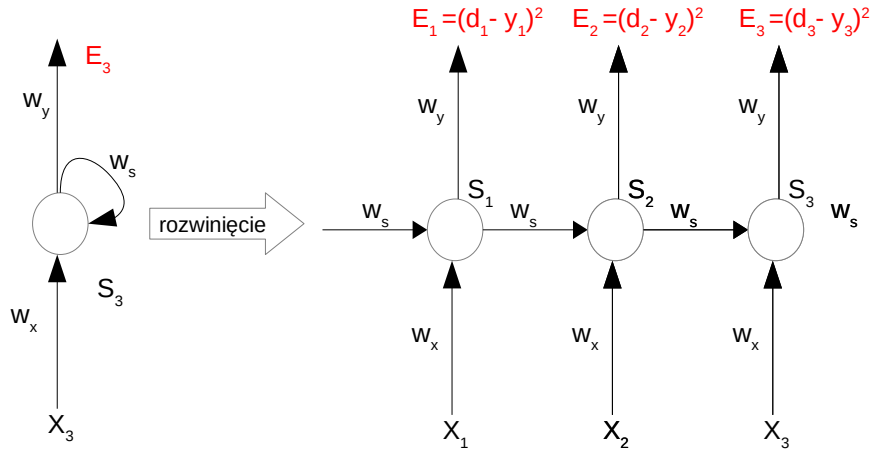
Popularną odmianą LSTM jest **Gated recurrent unit (GRU)**, w której połączono bramkę zapominającą z bramką aktualizacji przez co jedna jednostka bramkowa jednocześnie kontroluje proces zapominania oraz aktualizowania stanu. Dodatkowo połączony jest stan komórki wraz ze stanem ukrytym. W związku z tym uzyskany model jest prostszy obliczeniowo oraz w implementacji.



Rysunek 2.9. Struktura LSTM

2.2.2.2. Sieci RNN i propagacja wsteczna

W sieciach rekurencyjnych podczas treningu wykorzystywana jest wsteczna propagacja w czasie (ang. *Backpropagation Through Time - BPTT*). Algorytm ten bazuje na wyżej opisanej metodzie spadających gradientów, gdzie wykorzystywany jest łańcuch reguł wyliczania ich na podstawie struktury połączeń w sieci. Niektóre gradienty błędów przepływają wstecz z przyszłych do bieżących kroków czasowych, a nie tylko z kolejnych warstw, jak to ma miejsce w przypadku standardowej wstecznej propagacji. W sieciach RNN obecny stan zależy od wejścia oraz od stanu poprzedniego, dlatego też, aby zaktualizować wagi, należy wziąć pod uwagę gradient funkcji kosztu ze stanu obecnego obliczonego dla wejścia oraz gradienty z wszystkich wcześniejszych kroków czasowych. W związku z tym algorytm BPTT rozwija sieć neuronową w czasie, oblicza i akumuluje błąd dla każdego z nich, zwija sieć z powrotem, a na końcu aktualizuje wagi. Rozwiniętą sieć rekurencyjną obrazuje rysunek 2.10. Posługując



Rysunek 2.10. Rozwinięta w czasie sieć RNN

się oznaczeniami przyjętymi na rysunku 2.10, funkcja straty w czasie t może być zapisana jako:

$$E_t = (d_t - y_t)^2 \tag{2.28}$$

Wagi W_x, W_y, W_s reprezentują kolejno macierze wag połączeń <wejście, stan>, <stan, wyjście> oraz <poprzedni_stan, stan>. Obliczenie wartości gradientu dla W_y w trzecim kroku czasowym odbywa się poprzez:

$$\frac{\partial E_3}{\partial W_y} = \frac{\partial E_3}{\partial y3}\frac{\partial y3}{\partial W_y} \tag{2.29}$$

Obliczanie W_s zależy od stanu s_3, który zależy od stanu s_2, który z kolei zależy od poprzedzającego go stanu s_1. Stąd, wszystkie gradienty biorą udział podczas aktualizacji W_s:

$$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial y3}\frac{\partial y3}{\partial s_3}\frac{\partial s_3}{\partial W_s} + \frac{\partial E_3}{\partial y3}\frac{\partial y3}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial W_s} + \frac{\partial E_3}{\partial y3}\frac{\partial y3}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial W_s} \tag{2.30}$$

W ten sam sposób aktualizowane są wartości wag W_x. Generalnie formułę na aktualizacje wag można zapisać jako:

$$\frac{\partial y}{\partial W} = \sum_{i=t+1}^{t+N} \frac{\partial y}{\partial s_{t+N}}\frac{\partial s_{t+N}}{\partial s_i}\frac{\partial s_i}{\partial W} \tag{2.31}$$

Z powyższych zależności wysuwa się wniosek, iż trening sieci rekurencyjnej jest bardzo kosztowny obliczeniowo. Obliczenie gradientu dla sekwencji o długości *N* kroków jest tak samo kosztowny jak wytrenowanie sieci składającej się z *N* warstw. Stąd stosuje się przyciętą wersję BPTT zwaną TBPTT (ang. *truncated back propagation through time*), która zakłada częstsze aktualizacje wag. Jeżeli mamy *N* kroków, wówczas trening oparty na standardowym BPTT dla każdej osobnej aktualizacji parametrów, wymaga uwzględnienia *N* kroków czasowych w każdym przebiegu w przód i wstecz. Z tego względu złożoność obliczeniowa procesu rośnie bardzo szybko. W przyciętej wersji propagacji w czasie przebiegi w przód i wstecz są podzielone na mniejsze fragmenty, na których odbywają się aktualizacje. Wielkość kroku jest konfigurowanym parametrem.

3. Platformy sprzętowe

Poniższy rozdział zawiera opis architektur akceleratorów sprzętowych użytych w niniejszej pracy. Najważniejszymi z nich są karty graficzne, inaczej nazywane układami GPGPU (ang. *general-purpose computing on graphics processing units*), specjalizujące się w obliczeniach wysoce równoległych. Najbardziej efektywnymi obecnie układami GPGPU są układy z rodziny Nvidia i właśnie one zostały użyte jako akceleratory sprzętowe do zaproponowanych na potrzeby niniejszej rozprawy algorytmów. Oprócz opisu architektury układów GPGPU, w rozdziale tym zawarto również opis środowiska, najczęściej używanego do ich programowania tj. *CUDA*, oraz istotne cechy kart graficznych, na które należy zwrócić uwagę podczas ich programowania tak by zaimplementowane programy działały jak najbardziej optymalnie. Drugim akceleratorem sprzętowym użytym w tej pracy są standardowe procesory wielordzeniowe, które są głównie wykorzystywane do porównania efektywności algorytmów działających na kartach graficznych z ich odpowiednikami na procesorach ogólnego przeznaczenia, stąd też nie będą one opisywane.

3.1. Platformy sprzętowe a obliczenia równoległe

Projektując algorytm równoległy, najważniejszą częścią jaką należy wziąć pod uwagę jest docelowa platforma na której będzie on implementowany. Co więcej, implementując algorytm w akceleratorach takich jak układy GPGPU, należy mieć na uwadze konkretny model, na którym algorytm będzie działał, co jest szczególnie spowodowane różnymi możliwościami pamięciowymi jakie dana karta posiada. Wiodącą cechą akceleratorów sprzętowych jest umożliwienie przeprowadzenia obliczeń w sposób równoległy na co pozwala ich architektura. Generalnie równoległość może być zaimplementowana na następujących poziomach:

– równoległość na poziomie danych(ang. *data level parallelism - DLP*) - w tym podejściu jednocześnie przetwarzanych jest wiele różnych danych, np. w operacjach arytmetycznych,

– równoległość na poziome instrukcji (ang. *instruction level parallelism - ILP*) - tutaj jednocześnie wykonywanych jest więcej niż jedna instrukcja. Przykładami takiej równoległości są operacje *unrolling* wykonywane w układach GPGPU lub FPGA, bądź *potokowość* szczególnie popularana w układach FPGA,

– równoległość na poziome wątków (ang. *thread level parallelism - TLP*) - równoległość ta polega na wykonywaniu jednocześnie wielu wątków programowych w procesorze bądź na karcie graficznej,

– równoległość na poziomie procesów (ang. *process level parallelism - PLP*) - podejście to polega na wykonywaniu wielu programów na platformie wieloprocesorowej.

W literaturze [24] oraz [45] można spotkać inną taksonomię skupiającą się na powiązaniach między danymi oraz instrukcjami w systemach obliczeń równoległych, która przedstawia się on następująco:

– Single Instruction stream Single Data stream (SISD) – pojedyncza jednostka obliczeniowa CPU przetwarza pojedynczą instrukcję na danych przechowywanych w pojedynczej pamięci,

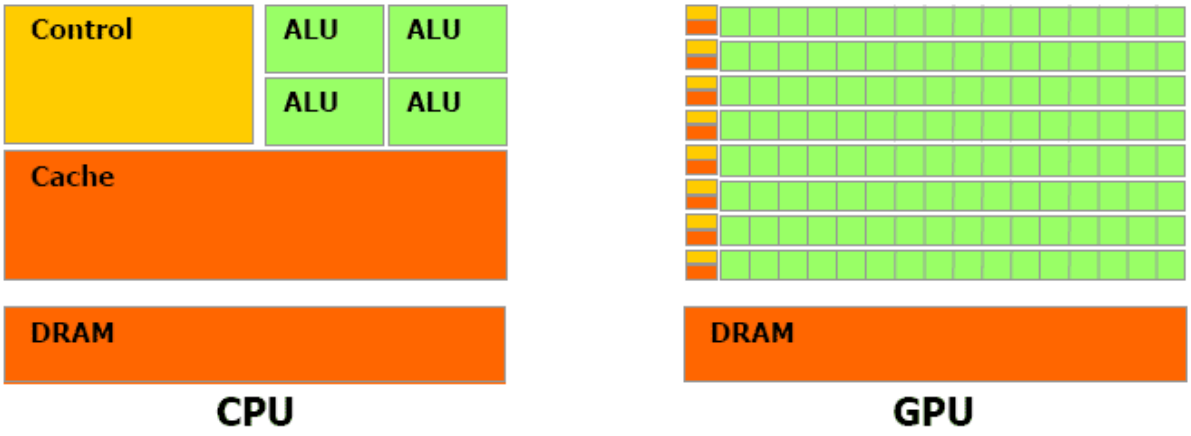
– Single Instruction stream Multiple Data stream (SIMD) – typowa architektura dla kart graficznych, w której wszystkie procesory wykonują te same instrukcje na różnych danych przechowywanych w lokalnej pamięci. Wymiana danych między procesorami odbywa się np. przez pamięć współdzieloną.

– Multiple Instructions stream Single Data stream (MISD) – wiele jednostek obliczeniowych operuje na tych samych danych, wykonując na nich różne obliczenia,

– Multiple Instructions stream Multiple Data stream (MIMD) – równoległe przetwarzanie zachodzi zarówno na poziomie danych jak i instrukcji.

3.2. Układy GPGPU

Budowa kart graficznych umożliwia przeprowadzenie obliczeń zgodnie z wyżej opisaną architekturą SIMD, która specjalnie dla kart graficznych, w celu lepszego zarządzania wątkami, została zmodyfikowana tak aby pojedyncza instrukcja była wykonywana przez wiele wątków, a wersja ta nosi nazwę SIMT (ang. Single-Instruction Multiple-Thread). Procesory graficzne są przeznaczone do przeprowadzania obliczeń silnie równoległych. Jest to możliwe poprzez umiejscowienie w nich większej liczby jednostek obliczeniowych kosztem modułów sterujących tj. buforowaniem danych oraz kontrolą przepływu instrukcji (rys. 3.1), co w przypadku obliczeń zgodnych z architekturą SIMT, prowadzi do minimalizacji opóźnień związanych z dostępem do danych, co zwiększa możliwości równoległego przetwarzania.



Rysunek 3.1. Porównanie budowy procesora oraz karty graficznej [111]

Schemat architektury wewnętrznej karty graficznej został zaprezentowany na 3.2. Najbardziej elementarną jednostką w procesorach tego typu jest *Streaming Processor (SM)*, których liczba jest zależna od konkretnego modelu karty. W skład budowy pojedynczego SM wchodzą:

– Stream processor (SP) - jednostka obliczeniowa, wykonująca obliczenia pojedynczej precyzji w jednym multiprocesorze,

– DP - jednostka obliczeniowa podwójnej precyzji,

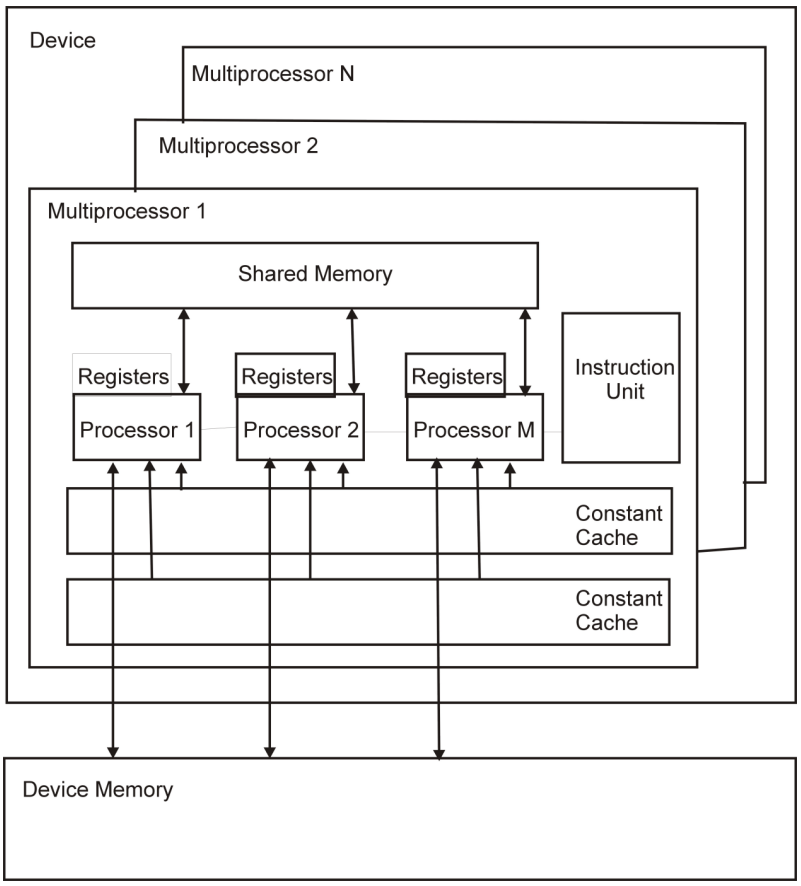
– SFU - jednostka przeznaczona do obliczeń bardziej skomplikowanych funkcji matematycznych, takich jak funkcje trygonometryczne bądź logarytmiczne,

– MT Issue - jednostka rozdzielająca zadania dla SP i SFU,

– C-Cache - bufor stałych, umożliwiający szybszy odczyt z obszaru pamięci stałych,

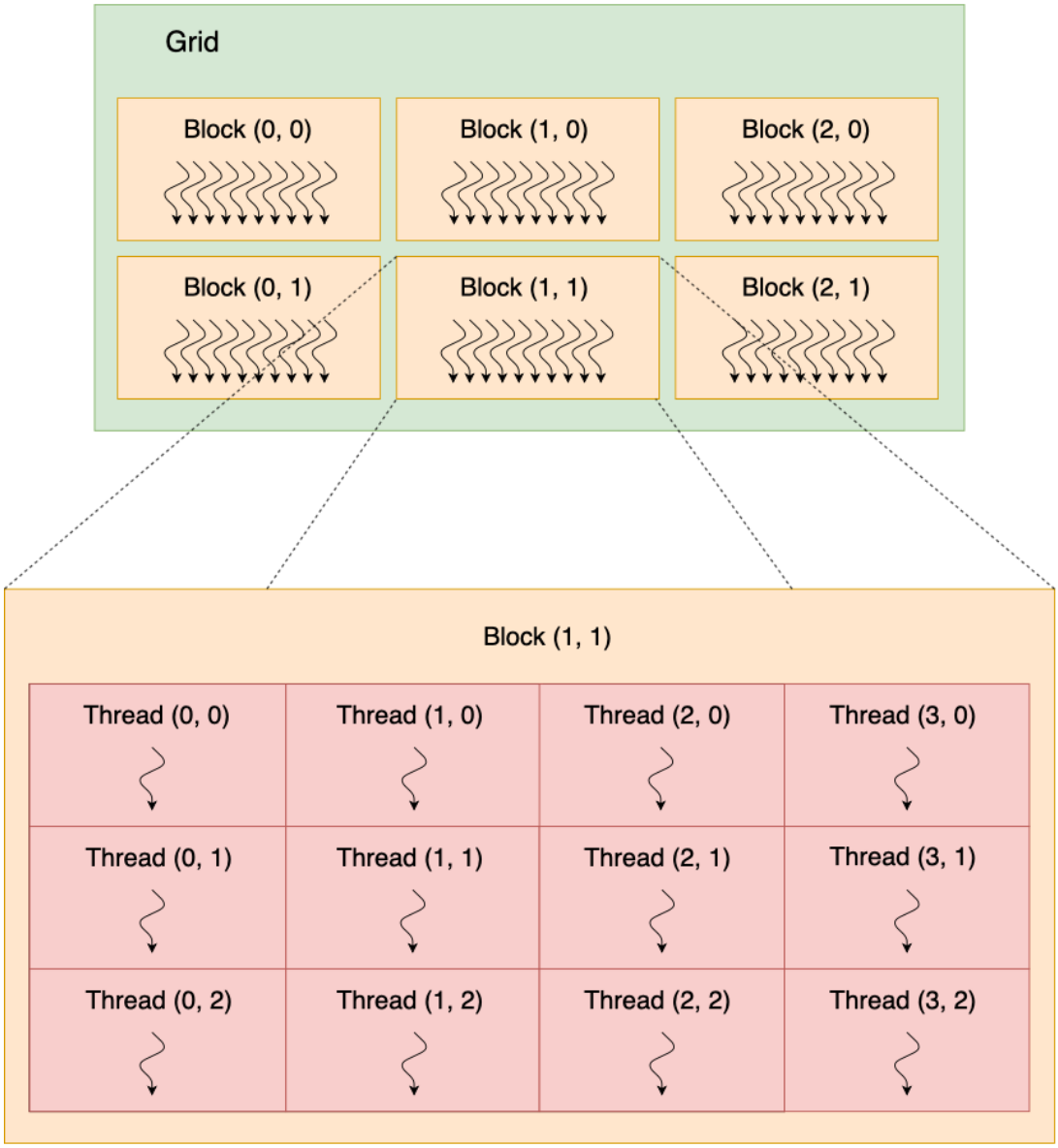
– I-Cache - bufor instrukcji,

– Shared memory - pamięć współdzielona, do której mają dostęp wątki w ramach jednego bloku.



Rysunek 3.2. Schemat architektury wewnętrznej karty graficznej

Multiprocesor jest odpowiedzialny za tworzenie, uruchamianie oraz zarządzanie wątkami. Dodatkowo dostarcza do nich dane, poprzez mechanizm umożliwiający ich niezależne pobieranie do jednostek obliczeniowych. Synchronizacja wątków na karcie graficznej jest operacją wymagającą tylko jednej instrukcji procesora, dzięki czemu możliwe jest dzielenie zadań na poszczególne wątki, a co za tym idzie umożliwia uzyskiwanie wysokiego poziomu równoległości.
Zarządzanie wątkami przez multiprocesor odbywa się poprzez tzw. *warpy*, czyli grupy 32 takich wątków. Aby program działał efektywnie należy zapewnić by wszystkie wątki w obrębie jednego warpa wykonywały te same instrukcje, gdyż w sytuacji wystąpienia instrukcji warunkowej, wszystkie wątki z warpa wykonają instrukcje z wszystkich rozgałęzień co spowoduje zmniejszenie efektywności wykonywania algorytmu. W obrębie jednego warpa nie jest wymagana synchronizacja. Warpy grupowane są z kolei w bloki wątków, posiadające dostęp do wspólnej pamięci współdzielonej. Zbiór tak utworzonych bloków nazywany jest gridem, co zostało pokazane na rysunku 3.3.



Rysunek 3.3. Schemat grupowania wątków na karcie graficznej

W układzie GPGPU możemy wyróżnić następujące rodzaje pamięci[1] :

– rejestry - każdy multiprocesor posiada 65536 32-bitowych rejestrów,

– pamięć współdzielona - mają do niej dostęp równoległy wątki w obrębie jednego bloku wątków - 48 [KB],

– pamięć stała - pamięć buforowana, służąca tylko do odczytu - 64 [KB]

– pamięć tekstur - podobnie jak pamięć stała służy tylko do odczytu i jest buforowana. Koszt odczytu rośnie, gdy potrzebna wartość nie znajduje się w buforze, wówczas dane wyciągane są z pamięci globalnej. Ten rodzaj pamięci jest optymalizowany pod kątem odczytywania bliskich sobie adresów, więc warto ją wykorzystać, gdy wątki w obrębie jednego warpa chcą odczytać dane spod adresów znajdujących się blisko siebie - 131 [KB] ,

– pamięć globalna - nie jest buforowana, dostęp do niej nie jest efektywny. Przez nią odbywa się wymiana danych między procesorem a kartą graficzną - 32510 [MB].

Z dostępem do pamięci globalnej wiąże się jeden ważny aspekt, mianowicie należy zadbać o tzw. *coalesced memory access*. Jest to związane z wcześniej wspomnianym grupowaniem wątków w warpy, co jest istotne nie tylko z punktu widzenia obliczeń, ale również dostępu do pamięci. W najnowszych architekturach GPGPU, dostęp do pamięci globalnej odbywa się sposób równoległy poprzez 32 wątki z jednego warpa, co jest nazwane pojedynczą transakcją (w starszych architekturach w pojedynczej transakcji brało udział 16 wątków czyli tylko pół warpa). Aby dostęp w obrębie transakcji był efektywny należy zapewnić, aby wątki o kolejnych indeksach wyciągały dane z kolejnych adresów pamięci (gdy wątek o indeksie 0 czyta spod adresu n wtedy wątek o indeksie 1 powinien czytać spod adresu $n+1$, a wątek o indeksie 31 spod adresu $n+31$). Dodatkowo należy zadbać o minimalizowanie liczby koniecznych do wykonania transakcji, co można osiągnąć poprzez tzw. *aligned data access*. Tablice w pamięci karty przechowywane są w postaci 256-bajtowych segmentów. Dostęp do segmentów może odbywać się w transakcjach, które wyciągają paczki o rozmiarach 32, 64 bądź 128 bajtów. W momencie gdy n jest początkowym adresem, którego wartość jest wielokrotnością 32, a pojedyncza transakcja wczytuje 32-bajty danych, wówczas do wczytania 32-bajtów wystarczy jedna transakcja. W momencie gdy program zaczyta 32 bajty danych startując od pozycji $n+1$ do adresu $n+32$ wówczas do wczytania danych, będą konieczne dwie transakcje, gdzie pierwsza wczyta dane z adresów od $n+1$ do $n+31$, po czym uruchomiona zostanie druga wczytująca ostatnią brakującą daną spod adresu $n+32$.

3.2.1. Karty graficzne Nvidia VOLTA

Jak zostało wcześniej wspomniane akceleratorem sprzętowym użytym w eksperymentach przeprowadzonych w niniejszej rozprawie jest karta graficzna typu **Nvidia Tesla V100-SXM2-32GB** [96]. W porównaniu do poprzednich architektur główną optymalizacją jest wprowadzenie w kartach tego typu,

[1] pojemności podanych pamięci mogą się różnić w zależności od typu karty. Wartości zaprezentowane poniżej, odpowiadają tym jakie posiada karta która została użyta do obliczeń, na potrzeby niniejszej pracy czyli: *:Tesla V100-SXM2-32GB* [96]

nowej generacji procesorów SM, które są o 50% [96] bardziej efektywne od tych użytych w ich poprzedniej wersji. Aby lepiej pokazać możliwości Tesli V100 załączono tabelę 3.4, pochodzącą ze strony producenta, zawierającą zestaw parametrów użytej karty w porównaniu z kartami poprzednich generacji.

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS[1]	5	6.8	10.6	15.7
Peak FP64 TFLOPS[1]	1.7	.21	5.3	7.8
Peak Tensor TFLOPS[1]	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm²	601 mm²	610 mm²	815 mm²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN
[1] Peak TFLOPS rates are based on GPU Boost Clock				

Rysunek 3.4. Porównanie parametrów kart NVIDIA Tesla [96]

Z punktu widzenia tej pracy najistotniejszym faktem jest w kartach tego typu zwiększanie możliwości akceleracyjnych w obliczeniach związanych ze sztuczną inteligencją. Najważniejszą modyfikacją jest wprowadzenie nowego typu rdzeni zwanych *Tenosr core*, dzięki którym możliwe staje się przyśpieszanie operacji macierzowych, które są sercem w procesie trenowania systemów sztucznej inteligencji. W Tesli V100 wprowadzono 640 takich rdzeni - 8 na każdy SM (tabela 3.4). Rdzenie tego typu stanowią programowalne jednostki nazywane przez producentów *matrix-multiply-and-accumulate units*, dzięki którym możliwe staje się dostarczenie do 125 TFLOPS (jednostka mocy obliczeniowej komputerów, definiująca możliwość wykonania operacji zmiennoprzecinkowych na sekundę) więcej mocy obliczeniowej, wy-

korzystywanej podczas treningu modelu sztucznej inteligencji np. sieci neuronowej. Każdy taki rdzeń procesuje macierz o rozmiarach $4 \times 4 \times 4$ wykonując na niej operacje $D = A * B + C$, gdzie wszystkie te macierze są rozmiarów 4×4. Aby obliczenia te mogły być wykonane się na rdzeniach *Tensor* dane wejściowe muszą być reprezentowane przez format $fp16(halfprecision)$ bądź int_8, czyli inaczej mówiąc musi zostać użyta zredukowana precyzja. Wynikiem operacji mnożenia i dodawania może być macierz przechowująca dane typu float bądź fp16. Każdy *tensor core* wykonuje w jednym cyklu zegara 64 akumulujących operacji mnożenia (FMA) na danych ze zredukowaną precyzją [96]. W związku z tym jeden SM, który zawiera 8 takich rdzeni wykonuje 1024 takie operacje w jednym takcie zegara, co 8-krotnie zwiększa wydajność aplikacji związanych z uczeniem głębokim w stosunku do kart typu Pascal GP100 [96].

3.2.2. Środowisko CUDA

Środowisko CUDA (ang. *Compute Unified Device Architecture*) zostało opracowane przez firmę NVIDIA i jest używane do programowania procesorów graficznych wyprodukowanych przez tę firmę. CUDA jest zintegrowana z językiem C/C++ zawierając przy tym definicję modelu programowania równoległego wraz z zestawem konkretnych instrukcji niskopoziomowych dla układu GPGPU.
Programowanie w języku CUDA, oparte jest na wywoływaniu z procesora (zwanego *hostem*), funkcji jąder programowych (zwanych *kernelami*), działających na karcie graficznej (zwanej *device*), które operują na strumieniach danych będących typu wektorowego. Funkcja na procesorze graficznym uruchamia się dla gridu, którego składowymi są bloki, z których każdy dzieli się na wątki (pojęcie gridu i bloku zostało wprowadzone we wcześniejszym podrozdziale). Każdy wątek posiada swoją lokalną pamięć oraz posiada dostęp do pamięci współdzielonej w ramach bloku. w którym się znajduje. Pamięć współdzielona służy do komunikacji wątków w obrębie bloku. W najnowszych architekturach kart graficznych (od architektury KEPLER [15]) przy użyciu wersji CUDA wyższej niż 7.5, możliwa jest, przy pomocy funkcji *__shfl*, bezpośrednia wymiana informacji między rejestrami wątków w obrębie jednego warpa, co jest bardziej efektywne niż uczynienie tego przy użyciu pamięci współdzielonej.
Na programiście spoczywa odpowiedzialność za ustalenie liczby wątków w obrębie bloku (zmienna *gridDim*) oraz za ustalenie liczby bloków działających w gridzie (zmienna *blockDim*). Obie te wartości są typu *dim3*, co oznacza że zarówno liczba bloków jak i wątków w pojedynczym bloku może być definiowana w postaci jedno-, dwu- bądź trój-wymiarowej. Identyfikacja wątku w bloku odbywa się poprzez zmienną *threadIdx*, natomiast blok w girdzie definiowany jest przy pomocy zmiennej *blockIdx*. Obie te zmienne są trój-wymiarowe, a dostęp do każdego wymiaru odbywa się poprzez użycie kolejno rozszerzeń *.x, .y* bądź *.z*. Globalny identyfikator wątku w obrębie gridu może być osiągnięty poprzez formułę: *i = blockIdx.x*blockDim + threadIdx.x* (w wymiarach *y* oraz *z*, globalne id jest pobierane analogicznie). Maksymalna liczba wątków w bloku, w zależności od karty wynosi 512 lub w nowszych modelach 1024. Liczba możliwe działających równolegle bloków jest zależna od liczby dostępnych SMs i może być odczytana z dokumentacji karty.
W środowisku CUDA komunikacja między *hostem* a *device* odbywa się poprzez pamięć operacyjną oraz pamięć globalną karty. Na programiście spoczywa rezerwacja odpowiedniej ilości pamięci, wypełnieniu

jej danymi oraz przesłaniu ich pomiędzy urządzeniami przy użyciu odpowiednich funkcji CUDA. Synchronizacja wątków może odbywać się za pomocą funkcji *__synchthreads()* bądź funkcji pochodzących z *Cooperative Groups API* [16], które zostało wprowadzane od wersji 9.0 i zawiera szereg funkcji umożliwiających komunikację między wątkami. Synchronizację można również zrealizować poprzez operacje atomowe oraz używając pamięci globalnej, jednak ten sposób jest najmniej efektywny i z tego względu nie jest zalecany. Programista nie ma wpływu na kolejność wykonywania bloków w gridzie oraz na ich synchronizację.

4. Równoległa implementacja algorytmów optymalizacji problemu *Low autocorrelation binary sequence* w układach GPGPU

W rozdziale tym zostanie przedstawiony efektywny sposób implementacji wybranych algorytmów heurystycznych w układach GPGPU, w celu lokalnej optymalizacji problemu *Low autocorrelation binary sequence (LABS)* [33]. Przez pojęcie algorytmu heurystycznego należy rozumieć algorytm najczęściej ze złożonością wielomianową, zwracający rozwiązanie przybliżone do najlepszego, w praktyce działający bardzo szybko. Metody te nie gwarantują znalezienia optymalnego rozwiązania problemu, jednak dążą one do wygenerowania rozwiązania dopuszczalnego, gdzie wartość funkcji celu jest maksymalnie zbliżona do optymalnej. Stosuje się je do rozwiązania problemów, dla których użycie algorytmów dokładnych takich jak przeszukiwanie wyczerpujące (ang. *brut force*) lub metoda podziel i ogranicz (ang. *branch and bound*) byłoby zbyt kosztowne ze względu na zbyt wysoką złożoność obliczeniową rozwiązywanego problemu. Warto wspomnieć, że metody heurystyczne można generalnie podzielić na dwie grupy:

– *heurystyki konstruktywne* - konstruujące rozwiązania krok po kroku, zwracający jedyne rozwiązanie wraz z zakończeniem procedury. Przykładami tego rodzaju heurystyki są np. algorytmy zachłanne, przeszukiwania wiązkowego oraz metody dekompozycji problemu,

– *heurystyki polepszające* - rozpoczynające działanie od rozwiązania losowego i w kolejnych krokach modyfikujące je, starając się uzyskać najlepsze. Przykładami takiej heurystyki są metaheurystyki czy też przeszukiwania lokalne, do których należą zaproponowane w niniejszym rozdziale metody, stąd też stanie się ono przedmiotem dalszej dyskusji.

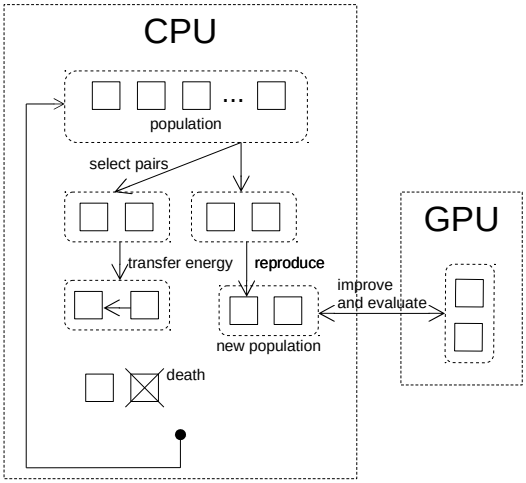
Generalnie rzecz ujmując heurystyki lokalnego przeszukiwania (ang.*local search*), opierają się na iteracyjnej poprawie aktualnego rozwiązania, gdzie w każdej iteracji wybierane jest dozwolone (w niektórych algorytmach niektóre ruchy są blokowane) rozwiązanie z sąsiedztwa, które posiada lepszą wartość funkcji celu od bieżącego. Rozwiązanie początkowe dla tego rodzaju heurystyk zazwyczaj generowane jest losowo, natomiast metoda kończy działanie w momencie nie wygenerowania wśród sąsiadów lepszego rozwiązania od bieżącego. Warto nadmienić, że w celu zapewnienia wielomianowej złożoności obliczeniowej w niektórych przypadkach występuje konieczność wprowadzenia dodatkowego warunku stopu, którym może być maksymalna liczba iteracji, bądź maksymalny czas przez który algorytm winien działać. W przypadku algorytmów, które blokują niektóre ruchy w danej iteracji, wprowadzenie dodatkowego

warunku stopu powoduje sprawdzenie większej liczby rozwiązań, gdyż algorytm nie kończy działania w momencie braku poprawy rozwiązania w kolejnym kroku, ale przeszukuje dalej i możliwe jest w tych przypadkach, że takie rozwiązanie zostanie znalezione, co zostanie w dalszych podrozdziałach pokazane dla algorytmu TABU. Zaproponowanymi w tej pracy algorytmami heurystycznymi iteracyjnymi, wykorzystanymi do rozwiązania problemu LABS, są *The steepest descent local search (SDLS)* [2] oraz *Tabu serach* [30][32], które jak dowiedziono w [29] osiągają bardzo dobre rezultaty przy rozwiązywaniu problemu LABS (dokładny opis wymienionych algorytmów znajduję się w dalszej części rozdziału). Warto nadmienić, że istnieje również trzeci algorytm sprawdzający się przy optymalizacji LABS, nazywany *self-avoidng walk (SAW)* [6] jednak nie jest on tematem badań niniejszej pracy, więc nie będzie on w dalszej części poruszany, nie mniej jednak ze względu na swoją skuteczność jego implementacja w układach GPGPU, jest rozpatrywana pod kątem przyszłych prac autora.

Zaimplementowane algorytmy *SDLS* oraz *Tabu* stanowią część konceptu mementycznego, łączącego opisane w rozdziale 2.1 ewolucyjne algorytmy agentowe wraz z heurystycznymi algorytmami optymalizacji lokalnej. Przykładem opisanego połączenia jest doskonale sprawdzająca się przy rozwiązywaniu skompilowanych obliczeniowo problemów, takich jak np. Golomb ruler (OGR) [55, 56] czy badany w tej pracy problem LABS [83], koncepcja **EMAS**, która nie stanowi obszaru badań niniejszej pracy, a której dokładny opis znajduje się w przedstawionych pozycjach. Z punktu widzenia wykonanej implementacji najważniejszy jest koncept platformy hybrydowej EMAS [83], w której ewolucyjna część algorytmu wykonywana jest na procesorze, natomiast etap lokalnej optymalizacji w układach GPGPU, co obrazuje rysunek 4.1. Przeprowadzone badania dotyczą jedynie części lokalnej optymalizacji, więc zostanie ona bliżej przestawiona w kolejnych podrozdziałach, których najważniejszą a zarazem innowacyjną częścią jest implementacja wyżej wymienionych algorytmów *SDLS* oraz *Tabu serach*, w układach GPGPU. Warto również nadmienić, że w przeprowadzonych badaniach prócz nowej implementacji dotychczasowych algorytmów, wprowadzono modyfikacje, dzięki którym uzyskano nowe wersje algorytmu *SDLS*, które nazwano *SDLS-2* oraz *SDLS-przeszukiwanie w głąb*, których implementacja ze względu na szeroki obszar przeszukiwań, ma sens jedynie w wersji równoległej i została również dokładnie opisana w dalszej części rozdziału. Dodatkowo, należy mieć świadomość, że dla ciągów o długościach nieparzystych, istnieje rozwiązanie redukujące przestrzeń poszukiwań o połowę (z 2^L do $2^{L/2}$). Rozwiązanie to bazuje na asymetryczności ciągów nieparzystych. Zaproponowane w tej pracy metody rozwiązujące problem LABS, nie pozwalają jednak skorzystać z tej cechy. Stąd też w większości przypadków, poszukiwane są rozwiązania problemu dla ciągów parzystych, a rozwiązanie wykorzystujące asymetryczność ciągów nieparzystych jest brane pod uwagę jako przyszłe prace badawcze autora. Więcej na temat asymetryczności i związanych z tym korzyści można znaleźć w [5][7].

4.1. Opis problemu LABS

Low autocorrelation binary sequence (LABS) jest z punktu widzenia złożoności problemem należącym do rodziny problemów NP-trudnych, jednocześnie charakteryzujący się nieskomplikowaną definicją. Problem ten od 1960 roku jest przedmiotem badań dla fizyków oraz naukowców z dziedziny sztucznej inteligencji. Problem skupia się na poszukiwaniu binarnej sekwencji $S = \{s_0, s_1, \ldots, s_{L-1}\}$ o



Rysunek 4.1. Uproszczona schemat zaproponowanej hybrydowej architektury dla mementycznego algorytmu EMAS [83]

długości L, z wartościami należącymi do dwuelementowego zbioru $s_i \in \{-1, 1\}$, z minimalną wartością energii $E(S)$ definiowaną jako:

$$\begin{aligned} C_k(S) &= \sum_{i=0}^{L-k-1} s_i s_{i+k} \\ E(S) &= \sum_{k=1}^{L-1} C_k^2(S) \end{aligned} \tag{4.1}$$

MJ Golay wprowadził w [34], współczynnik zwany *merit factor*, który bliżej odzwierciedla zależność między wyżej zdefiniowaną energią, a długością sekwencji na podstawie, której energia była obliczona. Dzięki czemu oddaje on dobrze jakość sekwencji i jest opisywany poprzez następującą formułę:

$$F(S) = \frac{L^2}{2E(S)} \tag{4.2}$$

Obszar poszukiwań dla problemu z długością sekwencji L ma rozmiar 2^L [71], natomiast energia dla takiej sekwencji jest obliczana w czasie $O(L^2)$. Problem LABS nie posiada żadnych ograniczeń związanych z ułożeniem wartości, stąd też występuje pełna dowolność przy binarnej reprezentacji reprezentacji tabeli S. Głównym powodem wysokiej złożoności obliczeniowej omawianego problemu jest wysoka korelacja pomiędzy wszystkimi jego elementami. Pojedyncza zmiana wartości elementu $C_i(S)$ ma wpływ na wiele innych elementów $C_j(S)$ i może prowadzić do dużej zmiany w wartości finałowej energii.

Czas potrzebny na poszukiwanie sekwencji z minimalną wartością energii może zostać skrócony poprzez zastosowanie rozwiązania zaproponowanego w [29], gdzie do rozwiązania problemu wprowadzono pojęcie *sąsiedztwa*, co znaczy że rozwiązanie S dla sekwencji o długości L jest uzyskiwane poprzez zamianę dokładnie jednego symbolu w sekwencji wejściowej, następnie dla tak powstałego nowego ciągu następuje obliczenie energii, po czym następuje powrót do oryginalnej formy ciągu wejściowego, obliczenie energii po zamienianie wartości na przeciwną na kolejnej pozycji, znów powrót do pierwotnej wartości ciągu wejściowego, co jest powtarzane L razy. W związku z czym powstanie L nowych energii,

s_1s_2	s_2s_3	s_3s_4	s_4s_5
s_1s_3	s_2s_4	s_3s_5	
s_1s_4	s_2s_5		
s_1s_5			

$T(S)$

$s_1s_2 + s_2s_3 + s_3s_4 + s_4s_5$
$s_1s_3 + s_2s_4 + s_3s_5$
$s_1s_4 + s_2s_5$
s_1s_5

$C(S)$

Rysunek 4.2. Struktury danych służące do efektywnego obliczenia funkcji celu w sąsiedztwie

z których każda powstała po zmianie dokładnie jednego elementu sekwencji w stosunku do sekwencji oryginalnej. Opisany sposób można zapisać jako:

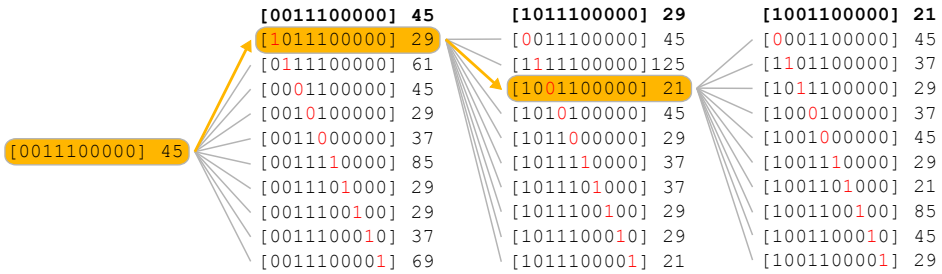
$$N(S) = \{flip(S, i)|i\{1, .., L\}\} \tag{4.3}$$

gdzie $flip(s_1...s_i...s_L, i) = s_1....s_i.....s_L$ [29].

Należy zauważyć, że zmiana jednej wartości oryginalnego ciągu nie musi prowadzić do obliczeń całej energii od nowa, gdyż jak wynika ze wzoru 4.1, nie wszystkie pośrednie iloczyny oraz sumy ulegają w takim przypadku zmianie. W związku z tym, wiele pośrednich wartości może swobodnie zostać reużytych prowadząc tym samym do zredukowania czasu potrzebnego do przeprowadzenia obliczeń. Aby było to możliwe powstałe poprzez zastosowanie formuły 4.3 wartości zostają zapisane w tabeli pomocniczej $T(S)$ o rozmiarach $(L-1) \times (L-1)$, takiej że $T(S)_{ij} = s_j s_{i+j}$ dla każdego $j \leq L-i$. Dodatkowo zostaje wprowadzona druga tabela pomocnicza o rozmiarze $L-1$ przechowująca korelacje konieczne do obliczenia ostatecznego rezultatu, definiowana jako $C_k(S)$ dla każdego $1 \leq k \leq L-1$. Rysunek 4.2 przedstawia wspomniane struktury dla sekwencji o długości $L = 5$. Autorzy rozwiązania zauważyli, że zmiana pojedynczego symbolu s_i, na przeciwny powoduje konieczność przemnożenia wszystkich komórek tabeli $T(S)$, w których występuje element o indeksie i, przez wartość -1. Dzięki temu spostrzeżeniu funkcja celu $flip(S, i)$ może być efektywnie obliczona w czasie $O(L)$ poprzez użycie algorytmu opisanego przez algorytm 1.

Algorytm 1. Obliczanie energii LABS przy użyciu struktur $T(S)$ oraz $C(S)$

```
1: function ValueFlip(S, i, T, C)
2:     f := 0
3:     for p := 0 to L − 1 do
4:         v := C_p
5:         if p ≤ L − i then
6:             v := 2T_pi
7:         if p < i then
8:             v := 2T_p(i−p)
9:         f := f + v^2
       return f
```



Rysunek 4.3. SDLS - iteracja po przykładowej sekwencji [83]

4.2. Opis algorytmu SDLS

Ogólna idea działania algorytmu SDLS dla problemu LABS polega na modyfikowaniu jednego bitu w wejściowej sekwencji po czym następuje sprawdzenie czy dla nowej sekwencji otrzymana energia jest niższa od poprzedniej jeśli tak, wówczas sekwencja, dla której ta energia została obliczona, ta staje się energią bazową, w której następuje zmiana losowego bitu i znów sprawdzenie czy nowa wartość jest lepsza od poprzedniej. Czynność ta jest powtarzana póki kolejne otrzymywane wartości są lepsze, w innym przypadku algorytm kończy działanie i zwraca aktualny ciąg jako najlepszy. Użycie SDLS do rozwiązania problemu LABS sposobem opisanym w podrozdziale 4.1, czyli do przedstawianej tam funkcji *flip*, polega na wybraniu najmniejszej spośród L wygenerowanych energii, po czym sprawdzeniu czy jej wartość jest niższa od wartości energii ciągu wejściowego. W przypadku spełniana tego warunku, ciąg dla którego została policzona optymalna energia, staje się ciągiem wejściowym do algorytmu 1. Kroki te powtarzane są tak długo, aż możliwe jest wygenerowanie lepszej energii. Działanie algorytmu zostało przedstawione za pomocą rysunku 4.3 oraz algorytmu 2. Należy podkreślić, że we wszystkich rysunkach ilustrujących pojedyncze iteracje (4.3, 4.4, 4.10, 4.11, 4.12), w celu zwiększenia ich czytelności, wartości -1 są reprezentowane przez wartość 0. Energie podane przy każdym ciągu są obliczane dla wartości realnych czyli -1.

4.3. Opis algorytmu *Tabu search*

Algorytm *Tabu search* był używany do rozwiązania problemu LABS [19][28], gdzie jego implementacja została wykonana na procesorach ogólnego przeznaczenia. Sama metoda może być postrzegana jako rozszerzenie wcześniej opisanego algorytmu SDLS o tak zwane *stany zabronione*. Algorytm wprowadza nową tablicę pomocniczą zwaną *tabu array*, o długości równej długości sekwencji, która zawiera zablokowane indeksy, które są kreowane w momencie, gdy lepsze rozwiązanie zostaje znalezione, wówczas indeks po zmianie, którego udało się tego dokonać, jest wstawiany do *tabu arrray*. Umiejscowienie danego indeksu w *tabu array* powoduje, że zostaje on zablokowany na M iteracji do przodu, co znaczy, że nie zostaje on zmieniony w celu poszukiwania energii. Głównym celem takiego postępowania jest próba wyjścia z lokalnego minimum, w jakim może znaleźć się algorytm SDLS, stąd też zablokowanie najlepszego indeksu na kilka iteracji (liczba iteracji na które zostaje zablokowany indeks jest parametrem konfigurowalnym) ma spowodować próbę poszukiwań optymalnego rozwiązania w innym obszarze

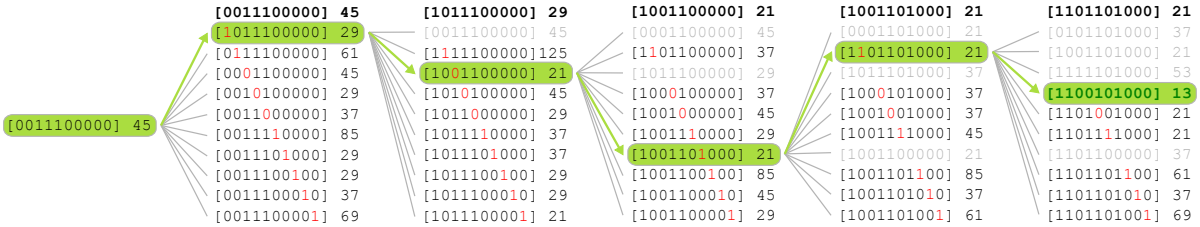
Algorytm 2. *Steepest Descent Local Search* dla problemu LABS [83], gdzie S – sekwencja wejściowa; L – długość sekwencji; F – ewaluacja sekwencji (*merit factor*); S_i/F_i – najlepsza sekwencja wraz z jej energią dla pojedynczej iteracji

```
1:  function SDLS(S)
2:      S_best = S
3:      F_best = EVALUATE(S_best)
4:      improvement = true
5:      while improvement do
6:          F_i = −∞
7:          for j = 0 to L − 1 do
8:              S_tmp = S_best
9:              S_tmp[j] = −1 ∗ S_tmp[j]
10:             F_tmp = EVALUATE(S_tmp)
11:             if F_tmp > F_cand then
12:                 S_i = S_tmp
13:                 F_i = F_tmp
14:         if F_i > F_best then
15:             S_best = S_i
16:             F_best = F_i
17:             improvement = true
18:         else
19:             improvement = false
20:     return S_best
```

(czyli po zmianie innego bitu na przeciwny). Działanie algorytmu zostało pokazane na rysunku 4.4, gdzie kolorem szarym zostały zaznaczone zablokowane przez mechanizm *Tabu* sekwencje. Można zauważyć, że wraz z kolejnymi iteracjami wyszarzane są kolejne sekwencje. Działanie rozszerzonej wersji SDLS o zabronione kroki, w sposób programistyczny zostało również zaprezentowane w pseudokodzie (algorytm 3). W takim przypadku konieczne jest ustawienie warunku stopu algorytmu na ustaloną liczbę iteracji, bądź czas w jakim algorytm powinien działać, gdyż brak poprawy w danym kroku nie znaczy w tym przypadku utknięcia algorytmu w lokalnym minimum, jak w takiej sytuacji dzieje się dla algorytmu SDLS bez TABU.

4.4. Równoległa implementacja algorytmu SDLS w celu lokalnej optymalizacji problemu LABS

Rozwiązanie problemu LABS przy pomocy algorytmu SDLS, może zostać zaimplementowane w sposób całkowicie równoległy, gdyż wyszukiwanie każdego z L rozwiązań jest niezależne od pozostałych $L-1$. Z tego też względu algorytm ten idealnie nadaje się do implementacji go w układach GPGPU.



Rysunek 4.4. Tabu - iteracja po przykładowej sekwencji (sekwencje zaznaczone na szaro nie mogą być wybrane, ponieważ zostały obliczone poprzez zmianę zablokowanego indeksu [83])

Algorytm 3. Algorytm *Tabu serach* dla problemu LABS [83], gdzie S – sekwencja wejściowa; L – długość sekwencji wejściowej; $maxIters$ – maksymalna liczba iteracji; F – ewaluacja sekwencji (*merit factor*)(; $tabu$ – wektor zawierający wartości typu *integer* opisujący na jak długo dany indeks pozostanie zablokowany; $minTabu$, $extraTabu$ – pomocnicze zmienne, wykorzystywne przy ustawianie indeksów w *tabu array*; $tabu$, $changedBit$ – inkdes po zmianie którego uzyskano najlepszą energię.

1: **function** TABUSEARCH($S, maxIters$)
2: $int[\,]\ tabu$ ▷ integer vector initialised with zeros
3: $minTabu = maxIters/10$
4: $extraTabu = maxIters/50$
5: $S_{start} = S$
6: $S_{best} = S$
7: $F_{best} = \text{EVALUATE}(S_{best})$
8: **for** $iteration = 0$ to $maxIters - 1$ **do**
9: $F_i = -\infty$
10: **for** $j = 0$ to $L - 1$ **do**
11: $S_{tmp} = S_{start}$
12: $S_{tmp}[j] = -1 * S_{tmp}[j]$
13: $F_{tmp} = \text{EVALUATE}(S_{tmp})$
14: **if** $iteration \geq tabu[j]$ **or** $F_{tmp} > F_{best}$ **then**
15: **if** $F_{tmp} > F_i$ **then**
16: $S_i = S_{tmp}$
17: $F_i = F_{tmp}$
18: $changedBit = j$
19: **if** $S_i \neq null$ **then** ▷ if better, non-blocked sequence is found
20: $S_{start} = S_i$ ▷ set the current sequence as the starting point for next iteration
21: $tabu[changedBit] = iteration + minTabu + \text{RAND}(0, extraTabu)$
22: **if** $F_i > S_{best}$ **then**
23: $S_{best} = S_i$
24: $F_{best} = F_i$
25: **return** S_{best}

Moduł odpowiedzialny za optymalizowanie problemu LABS używając metody SDLS, jako wejście dostaje ciąg binarny o długości L w postaci $\{-1, 1\}^L$, który może być wygenerowany poprzez algorytm genetyczny bądź w sposób losowy. Celem implementacji powstałej na potrzeby opisywanego rozdziału było dostarczenie optymalnej wersji odpowiedzialnej za implementację algorytmu przeszukiwania lokalnego stąd też sposób dostarczenia danych nie ma wpływu na szybkość obliczeń dlatego też nie będzie on przedmiotem dalszych dyskusji. Ważne jest, iż w celach optymalizacyjnych interfejs przyjmuje pewną określoną liczbę ciągów w wyżej przedstawionej postaci, w związku z tym zwiększają się szanse na znalezienie tego optymalnego, gdyż poszukiwania odbywają się na *K* ciągach wejściowych. Architektura karty pozwala na równoległe (oczywiście do pewnego zakresu - w tym przypadku, ograniczeniem jest maksymalna liczba bloków wątków, które mogą działać w sposób równoległy) przetwarzanie pewnej liczby ciągów wejściowych, gdzie każdy z nich z punktu widzenia algorytmu genetycznego jest osobnikiem, ponieważ zostają one rozrzucone pomiędzy bloki wątków. Niezwykle istotnym elementem jest zapewnienie różnorodności dla każdego z ciągów wejściowych, gdyż dostarczenie dwóch jednakowych ciągów wejściowych do dwóch różnych bloków spowoduje wygenerowanie identycznego rozwiązania (algorytm nie posiada żadnego kroku losowości). W związku z tym, każdy blok otrzymuje swój własny w miarę możliwości unikatowy binarny ciąg wejściowy S_I o długości L. Na jego podstawie budowane są dwie struktury pomocnicze (opisane bliżej w podrozdziale 4.1) $T(S)$ oraz $C(S)$ (rysunek 4.2), które są przechowywane w pamięci współdzielonej karty (ang. *shared memory*), a których dokładniejszy opis znajduję się w rozdziale 3.2. W celu stworzenia owych struktur, należy wykonać $L - 1$ iteracji. Podczas każdej z nich obliczany jest iloczyn $s_i s_{i+distance}$, gdzie $distance < L$. Podczas pierwszej iteracji współczynnik $distance$ ustawiony jest na 1, a liczba aktywnych wątków jest równa $L-1$. Po skończonej pierwszej iteracji, pierwszy rząd w strukturze $T(S)$ jest obliczony, a na jego podstawie używając operacji redukcji (zostanie ona bliżej opisana w dalszej części rozdziału), obliczany jest pierwszy rząd struktury $C(S)$. Następnie w każdej kolejnej iteracji inkrementowany jest, parametr $distance$ i dekrementowana jest liczba wątków, biorących udział w obliczaniu wartości, kolejnego rzędu struktury $T_{iteration}(S)$, na podstawie którego ta sama liczba wątków użyta jest do obliczenia kolejnego rządu $C_{iteration}(S)$ (jako sumy elementów rzędu $T_{iteration}(S)$). W celu obliczenia pierwszej energii, na podstawie wejściowego ciągu, która będzie używana jako energia referencyjna E_r, rezultaty każdej operacji redukcji dokonanej na elementach struktury $T_{iteration}(S)$ (czyli wartościach wstawianych do kolejnych rzędów struktury $C_{iteration}(S)$), są po każdej iteracji podnoszone do kwadratu po czym wynik dodawany jest do wartości uzyskanej w poprzedniej iteracji (dla pierwszej iteracji wartość z poprzedniej wynosi zero). Mając w pamięci zbudowane pomocnicze struktury $T(S)$ oraz $C(S)$ oraz obliczono energię referencyjną E_r, kolejnym krokiem jest przetwarzanie algorytmu SDLS poprzez wyszukiwanie lokalnego optimum używając algorytmu opisanego w rozdziale 4.2. Równoległa koncepcja jest podobna do sekwencyjnej, opisanej we wspominanym rozdziale, z tą różnicą że w tym przypadku mutacja każdego bitu zostaje dokonana równolegle przez osobny wątek, przez co równolegle w obrębie jednego bloku wątków, obliczanych jest L energii, z których każda powstała po zmianie jednego bitu w stosunku do energii wejściowej. Jak zostało wcześniej powiedziane równolegle działa K bloków więc na raz obliczanych jest $K \times L$ energii. Aby w obrębie bloku obliczyć L energii, po pierwsze tyle wątków musi zostać uruchomionych (aktualnie najdłuższe sekwencje są krótsze od maksymalnej liczby wątków jaka może być uruchomiona w obrębie

bloku), a każdy z nich wykonuje następujące operacje:

– zmiana wartości w sekwencji o indeksie $s_{\text{indexOfThread}}$, na przeciwną poprzez przemnożenie jej przez $-1(-1 * s_{\text{indexOfThread}})$ (Alg. 5, linia 6)

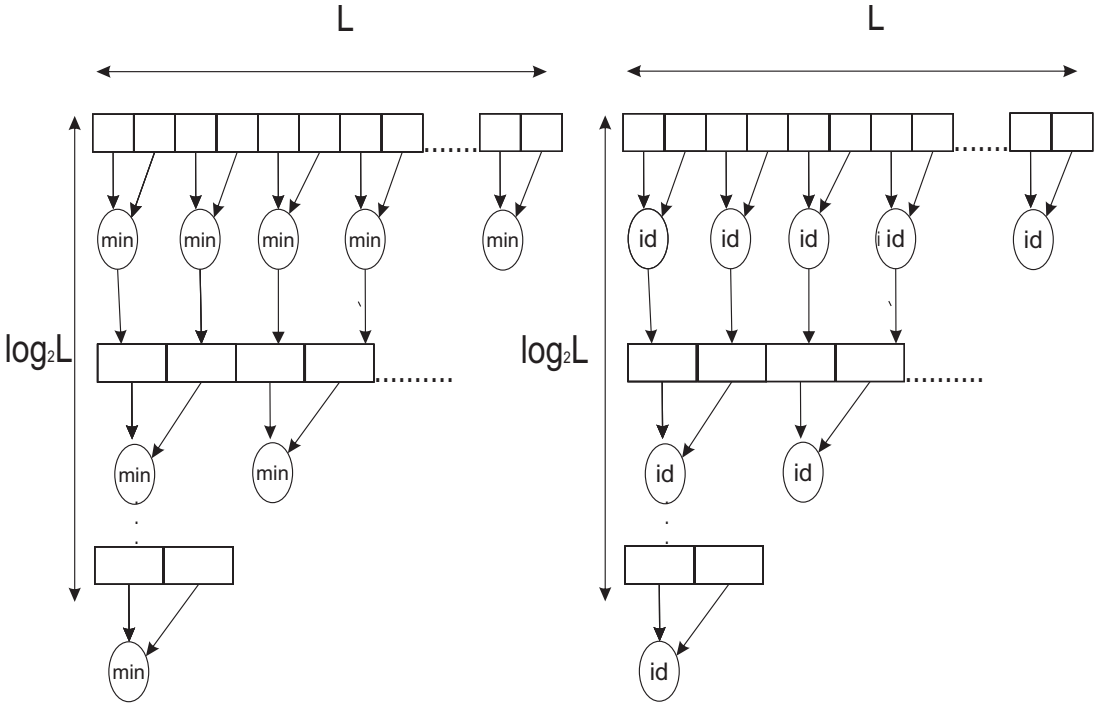
– obliczenie nowej wartości energii (po zmianie bitu) zgodnie z algorytmem *value flip* (Alg. 5, linia 7),

– zapisanie nowej energii w pamięci współdzielonej oraz w pamięci podręcznej wątku wartości $C'(S)$ w celu uniknięcia ponownej jej wyliczenia, gdyż *zwycięski* wątek musi uaktualnić swoimi wartościami dostępną dla wszystkich wątków strukturę $C(S)$ (Alg. 5, linia 9).

Zgodnie z funkcją $ValueFlip$ z algorytmu 4 zbudowaną na podstawie algorytmu 1, finalna energia f jest uzyskiwana poprzez zsumowanie wartości v, które to reprezentują komórki w pomocniczej strukturze danych $C(S)$. Mając wyliczonych L energii należy wybrać najmniejszą z nich. Aby tego dokonać uruchamiany jest proces podwójnej redukcji, który jest wykonywany przez L wątków, co obrazuje rysunek 4.5. W implementacji wersji redukcji, wykorzystywanych w tej pracy, czyli tej dokonującej sumowania elementów oraz tej odpowiedzialnej za wyszukiwanie najmniejszego z nich, użyta została funkcja *_shfl*, której użycie jest możliwe na używanej kartach nowszej generacji (wprowadzona została w architekturach *kepler*[112]), a dzięki której wątki pochodzące z jednego *warp* (rozdział 3.2) mogą dokonać operacji na danych zapisanych w ich pamięciach podręcznych, dzięki czemu wymiana informacji między wątkami jest szybsza od tej dokonanej przez pamięć współdzieloną. Pierwsza z redukcji dokonuje wyboru najlepszej energii, druga zwraca indeks wątku, który tę energię wyliczył. Informacja o indeksie *zwycięskiego* wątku jest konieczna do możliwości zwrócenia najlepszej sekwencji oraz do zaktualizowania przechowywanej w pamięci współdzielonej bloku struktury $C(S)$, gdyż w zaproponowanym podejściu każdy wątek tworzy w swojej pamięci podręcznej tablicę $C'(S)$, zbudowaną na podstawie wartości v. Tylko jeden wątek, który obliczył energię najlepszą w danej iteracji, dokonuje podmiany wartości $C(S)$ używając swoich wartości z $C'(S)$. Dodatkowo ten sam wątek dokonuje zaktualizowania wartości w strukturze $T(S)$, zmieniając tylko te komórki, które zawierają wartości do których wyliczenia był ten wątek używany (Alg. 5, linia 10). Najlepsza czyli minimalna energia E_{min} jest porównywana z energią referencyjną E_r obliczoną z sekwencji wejściowej. W sytuacji, gdy $E_{min} < E_r$, wówczas E_{min} staje się nową energią referencyjną, a sekwencja dzięki której została wyliczona nową sekwencją wejściową S_I (Alg. 5, linia 9). Pracę jaką wykonuje pojedynczy blok dodatkowo obrazuje rysunek 4.6.

Programując kartę graficzną, jedną z kluczowych ról odgrywa zarządzanie pamięcią. Stąd też należy być ostrożnym podczas jej alokacji. Zawsze trzeba brać pod uwagę układ na jakim dany program będzie uruchomiony, i sprawdzić że pamięć jaką dysponuje dany układ GPGPU jest wystarczająca aby zaalokować struktury danych konieczne by dany algorytm działał poprawnie. W przypadku algorytmu SDLS, rozwiązującego problem LABS, dla sekwencji o długości L, konieczne było zarezerwowanie pamięci współdzielonej (ang. *shared memory*) do przechowywania struktur danych, o następujących rozmiarach:

– $L * sizeof(short)$ – reprezentacja sekwencji,



Rysunek 4.5. Operacja redukcji wyszukująca najlepszą energię wraz z indeksem, pod którym się znajduje [129]

– $\frac{(L-1)*L}{2} * sizeof(short)$ – tablica $T(S)$

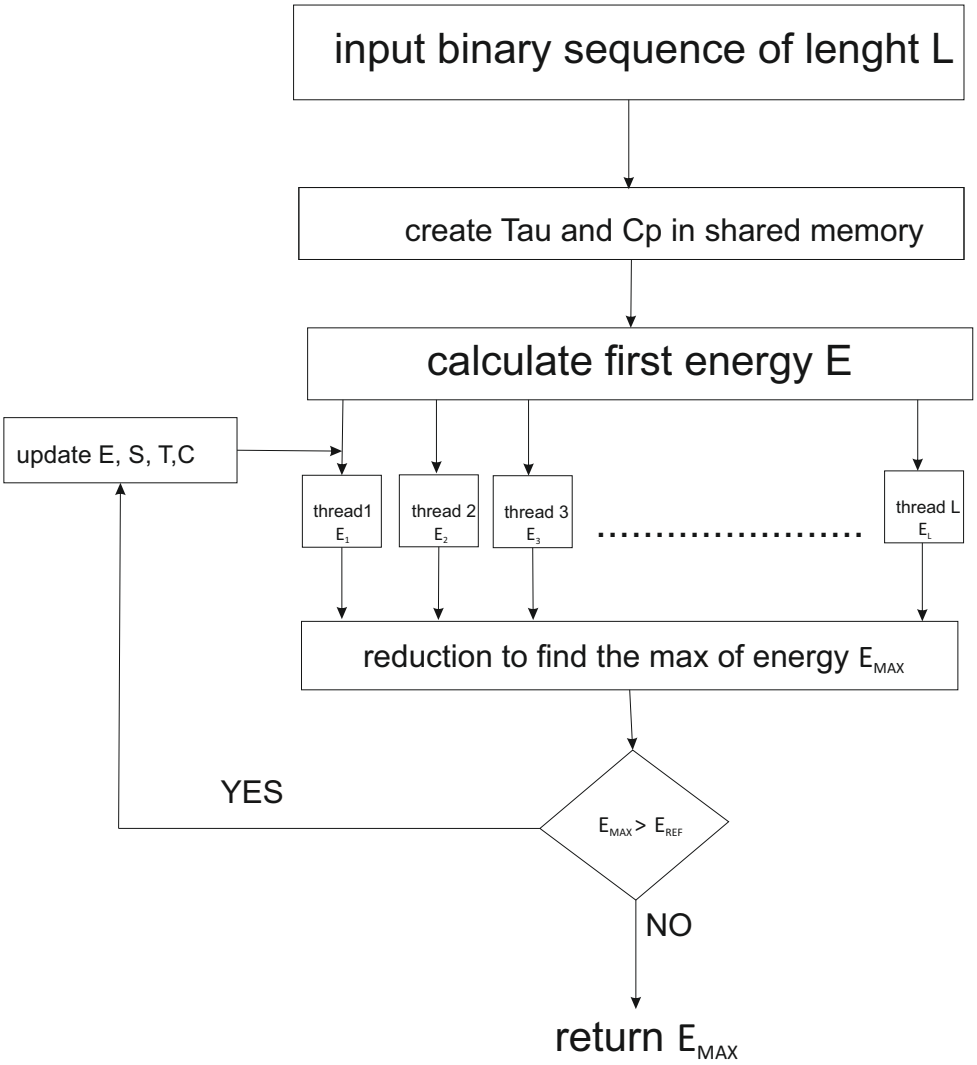
– $(L-1) * sizeof(int)$ – tablica $C(S)$,

– $2 * L * sizeof(int)$ – tablica przechowująca energię po kalkulacji przez każdy wątek wraz z odpowiadającymi im indeksami (na niej dokonywana jest operacja redukcji zwracające energię oraz id wątku, który ją obliczył).

Cotta [29] zaproponował, aby algorytm działał tak długo, aż niemożliwe stanie się polepszenie rezultatu w kolejnej iteracji, czyli jeśli zajdzie warunek: $E_{min} > E_r$ (dalsze iteracje będą zwracać ciągle ten sam wynik równy E_r). Wówczas blok powinien skończyć działanie i zwrócić najmniejszą energię,

Algorytm 4. Algorytm *Value flip* realizowany przez każdy wątek w GPGPU

```
1: function ParrallelValueFlip(S, i, T, C)
2:     f := 0
3:     for p := 0 to L − 1 do
4:         v := C_p
5:         if p ≤ L − i then v := 2T_{pthreadIndex} end if
6:         if p < i then v := 2T_{(threadIndex−p)} end if
7:         f := f + v^2
       end for
8:     return f
   end function
```



Rysunek 4.6. Poszukiwanie najlepszej energii realizowane przez pojedynczy blok wątków [83]

wraz z sekwencją dla której została ona znaleziona. Należy jednak pamiętać, że takich energii zostanie zwróconych K, gdyż tyle bloków zostało uruchomionych więc cały program skończy działanie w momencie, gdy ostatni blok znajdzie swoją optymalną energię, która niekoniecznie będzie energią najmniejszą wśród wszystkich zwróconych przez bloki energii. Dodatkowo należy podkreślić fakt, że dla jednakowej sekwencji wejściowej uruchomionej na procesorze niekoniecznie zostanie zwrócony ten sam rezultat co dla wersji uruchomionej w GPGPU, mimo że jak zostało wcześniej wspomniane, algorytm nie posiada kroku, w którym następowałoby losowanie. Spowodowane jest to sposobem wyboru najlepszej energii, który w obu przypadkach dokonywany jest przez operację redukcji, jednak nie jest pewne, że ta sama energia zostanie wybrana, w sytuacji gdy wystąpią na przykład dwie jednakowe wartości optymalne i wówczas nie zawsze oba układy wybiorą tę samą wartość. W sytuacji, gdy takie same, ale nie te same energie zostaną wybrane dla układu GPGPU oraz CPU, w kolejnej iteracji algorytmy otrzymają inne ciągi wejściowe, co może spowodować inną liczbę iteracji prowadzącą do znalezienia optymalnej wartości (do uzyskania warunku stopu). Dlatego też, w celu porównania szybkości działa wersji CPU z GPGPU, została ustalona stała liczba iteracji, mimo że w pewnym momencie w kolejnych

iteracjach nie będą uzyskiwane lepsze rezultaty, ale dzięki temu możliwe będzie dokonanie miarodajnego porównania czasowego między układem GPGPU a procesorem wielordzeniowym. Wersja na CPU została zaimplementowana w wersji równoległej dzięki bibliotece OpenMp [100]. Ostatnim krokiem jaki zostaje wykonany jest zwrócenie najlepszej energii spośród wszystkich energii zwróconych przez bloki, która zostaje znaleziona również dzięki użyciu operacji redukcji, gdzie w tym przypadku należy prócz wartości energii zwrócić indeks zwycięskiego bloku, by możliwe było odtworzenie najlepszego ciągu. Cały proces poszukiwania najlepszej energii dla układu GPGPU został zobrazowany na rysunku 4.7.

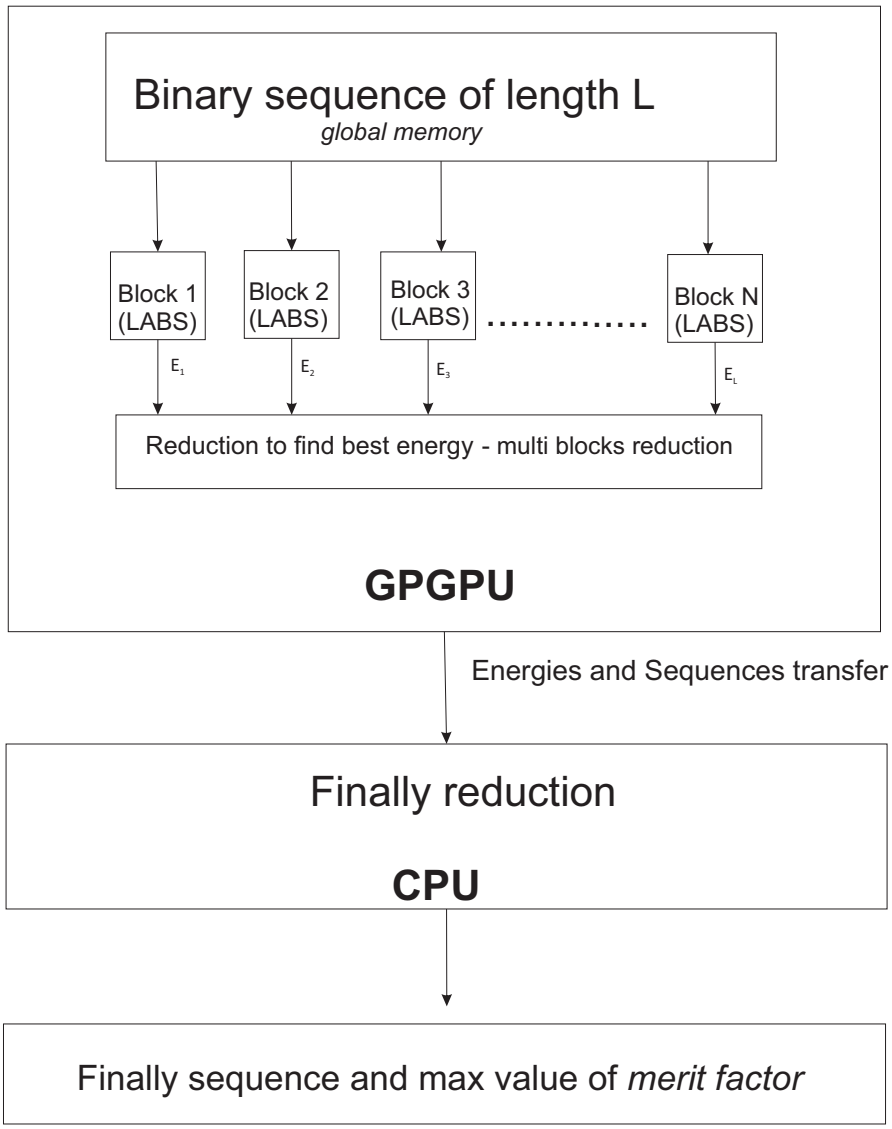
Algorytm 5. Równoległa wersja SDLS w GPGPU

```
1:  function PARALLELSDLS(S)
2:      transfer_solutions_to_GPU_blocks(S)
3:      for block := 0 to len(S) do
4:          compute_reference_energy(S_block)
5:          for iter := 0 to nr_of_iterations do
6:              mutation_of_threadId_bit(S_block)
7:              ParallelValueFlip(S_block, T', C')
8:              compute_lowest_energy()
9:              update_reference_energy()
10:             update_T(S)_and_C(S)()
            end for
        end for
11:     return f
    end function
```

Tabele 4.1 oraz 4.2 zawierają wyniki czasowe zarówno dla implementacji GPGPU jak również dla CPU, w których zostały pokazane przyśpieszania względem implementacji 1, 4 oraz 12 wątkowej dla przykładowych sekwencji o długościach $L = 48$ oraz L=201. Jak zostało wcześniej wspomniane zrównoleglenie na procesorze, zostało uzyskane dzięki bibliotece OpenMP. Najlepsze rezultaty na CPU, uzyskano gdy pozostawiono bibliotece dowolność przy wyborze liczby wątków, co jest realizowane poprzez użycie flagi *omp_set_dynamic(0)*. Dodatkowo, w eksperymentach w celach optymalizacyjnych, została użyta specjalna dyrektywa kompilatora *-O3*, która powoduje włączenie najwyższego stopnia optymalizacji wynikowego kodu maszynowego, wydłużając jednocześnie sam proces kompilacji. Dla obu zaproponowanych długości czasy zostały zmierzone dla różnej liczby osobników (32, 64, 128, 256), co w przypadku kart odzwierciedla liczbę uruchomionych bloków wątków. Każdy eksperyment został uruchomiony 10 razy, a zaprezentowane czasy są średnią arytmetyczną z czasów uzyskanych podczas każdego uruchomienia. W przypadku GPGPU obliczenia zostały uruchomione na karcie: *Nvidia Tesla V100-SXM2-32GB[96]* natomiast dla CPU na procesorze: *Intel(R) Core(TM) i7-9750H CPU(2.60GHz)*. Prócz wyników czasowych tabele 4.1 oraz 4.2, zawierają przyśpieszenie jakie zostało uzyskane dzięki implementacji w układach GPGPU, które zostało obliczone według formuły:

$$S_p = \frac{T_{CPU}}{T_{GPU}}. \tag{4.4}$$

Jak można zauważyć implementacja w GPGPGU jest za każdym razem efektywniejsza od implementacji na CPU, gdzie najmniejsze przyśpieszenie zostało uzyskane dla $L = 201$ i w porównaniu do urucho-



Rysunek 4.7. Cały proces poszukiwania najlepszej energii

mienia wersji CPU przy użyciu dwunastu wątków i wynosi ono $\sim 22.5 - 38.1$, w zależności od liczby osobników. Należy zauważyć, że przyśpieszenie wzrasta wraz ze wzrastającą liczbą osobników, gdyż w przypadku kart graficznych, każdemu osobnikowi odpowiada blok, więc im więcej osobników tym więcej zostaje uruchomionych nowych bloków, które wykonują się równolegle. Warto nadmienić, iż ten sam algorytm dla przypadku $L = 201$, uruchomiony na karcie starszej generacji *NVIDIA Tesla m2090* jest wolniejszy ponad 5 razy, od tego uruchomionego na kartach z rodziny Volta.

4.5. Równoległa implementacja algorytm *Tabu search* w celu lokalnej optymalizacji problemu LABS

Algorytm tabu może zostać zaadoptowany jako część wielu optymalizacyjnych algorytmów. Jak zostało wcześniej powiedziane, jego zadaniem jest zablokowanie kilku przyszłych ruchów algorytmu, co jest realizowane przez przechowywanie ich w tablicy pomocniczej. W przypadku implementacji w ukła-

Tabela 4.1. Czas podany w milisekundach konieczny do wykonania algorytmu SDLS ze 128 iteracjami dla problemu LABS dla $L = 48$

L. rozwiązań	GPGPU	CPU (1∼th)	Akceleracja	CPU (4∼th)	Akceleracja	CPU (12∼th)	Akceleracja
32	0.26	41	157,7	18,1	69,7	11.1	42,1
64	0.31	72.3	233,2	30.1	97,1	21,4	69
128	0.44	165	375	50	113,6	36	81,1
256	0.58	322,4	586,1	89.4	154,1	59.6	102,8

Tabela 4.2. Czas podany w milisekundach konieczny do wykonania algorytmu SDLS ze 128 iteracjami dla problemu LABS dla $L = 201$

L. rozwiązań	GPGPU	CPU (1∼th)	Akceleracja	CPU (4∼th)	Akceleracja	CPU (12∼th)	Akceleracja
32	5.4	742	137,4	189	35	132	22,5
64	8.9	1497	168,3	360	40,4	290	32,5
128	16.5	2970	181,6	745	45,5	580	35,4
256	31.4	6008	191,3	1649	52,5	1192	38,1

dach GPGPU ma to ogromne znaczenie, gdyż wymusza konieczność za-alokowania dodatkowej pamięci, co zawsze może być problemem w przypadku programowania sprzętowego, ze względu na ograniczoną ilość dostępnej pamięci. W przypadku zastosowania zabronionych ruchów w algorytmie SDLS, wykorzystanego do rozwiązania problemu LABS, struktura *tabu array* przechowuje indeksy w sekwencji, które zostają zamrożone na kilka iteracji do przodu oraz numer iteracji, w której dany bit został w tej tablicy umieszczony. Zamrożenie indeksu oznacza, że zmiana znajdującej się pod nim wartości nie może doprowadzić do znalezienia najlepszego rozwiązania, nawet jeśli taka sytuacja miałaby miejsce. Aby lepiej przybliżyć opisywaną sytuację, należy rozważyć następujący przykład: jeśli w iteracji X najlepsza energia została obliczona po zmianie bitu B, wówczas ten bit zostaje zamrożony na M iteracji (gdzie wartość M oznacza głębokość *tabu memory*) i po $X + M$ iteracjach, zablokowany w X iteracji indeks B zostaje odmrożony i może brać udział w poszukiwaniach najlepszej energii, do momentu gdyby znów po jego zmianie nastąpiło wyliczenie energii optymalnej.

W przypadku implementacji równoległej w GPGPU, algorytm tabu (Alg. 6) rozszerza opisaną powyżej równoległą wersję algorytmu SDLS (rozdział 4.4 oraz Alg. 5), poprzez zablokowanie zwycięskiego wątku (w wersji równoległej identyfikator wątku jest równoważny z indeksem w sekwencji) na M ruchów do przodu, stąd też przestaje być on aktywny. Zależność między wątkami oraz blokami a sekwencjami LABS jest identyczna jak w przypadku algorytmu SDLS (rozdział 4.4). Praca jaką w każdej iteracji wykonuje wątek, w opisywanym rozwiązaniu, przedstawia się następująco:

– zmiana wartości w sekwencji o indeksie $s_{\text{indexOfThread}}$ na przeciwną,

– obliczenie nowej wartości energii (po zmianie bitu) zgodnie z algorytmem *value flip*,

– zapisanie nowej energii w pamięci współdzielonej oraz w pamięci podręcznej wątku wartości $C'(S)$,

– znalezienie najlepszej energii (poprzez operację redukcji),

– aktualizacja *tabu array* poprzez wpisanie nowych zamrożonych indeksów oraz zwolnienie indeksu w przypadku, gdy przebywał w *tabu array* przez M iteracji (Alg. 6 linie 11 oraz 12)

Ostatni krok jest wykonywany tylko przez wątek, dzięki któremu możliwe było uzyskanie najniższej wartości energii. Podobnie jak poprzednio, wątek ten dokonuje aktualizacji struktur $C(S)$ oraz $T(S)$. Tablica *tabu array* zapisana jest w pamięci podręcznej, a jej długość jest równa podwojonej długości wejściowej sekwencji, gdzie na pierwszych L pozycjach jest przechowywana w sposób binarny informacja o tym czy dany bit jest zablokowany, a na kolejnych L dodatkowa informacja na ile iteracji dany indeks jest zablokowany (w przypadku gdy indeks nie jest zablokowany wartość ta wynosi zero).

Algorytm 6. Równoległa wersja algorytmu Tabu w GPU

```
1:  function PARALLELTABU(S)
2:      transfer_solutions_to_GPU_blocks(S)
3:      for block := 0 to len(S) do
4:          compute_reference_energy(S_block)
5:          for iter := 0 to nr_of_iterations do
6:              mutation_of_threadId_bit(S_block)
7:              ParallelValueFlip(S_block, T', C')
8:              compute_lowest_energy()
9:              update_reference_energy()
10:             update_T(S)_and_C(S)()
11:             update_tabu_list(current_best_move)
12:             free_tabu_list(older_moves_than_M)
            end for
        end for
13:     return f
    end function
```

Przyśpieszenia względem implementacji w procesorze mają się podobnie jak te przedstawione w poprzedniej sekcji, gdzie porównano czas względem czystego SDLS. Dołożenie tablicy *tabu array* oraz jej aktualizacja nie ma dużego wpływu na finalne wyniki czasowe.

4.6. Pomiar skuteczności algorytmów SDLS oraz *Tabu search* dla problemu LABS

Aby sprawdzić skuteczność algorytmów SDLS oraz Tabu serach dla problemu LABS, oba algorytmy zostały uruchomione jako część koncepcji EMAS na hybrydowej platformie AgE [107], dostarczającej rozwiązanie agentowe. Sam program napisany jest w języku Java [103], a podłączenie do niego kodu napisanego w środowisku Cuda było możliwe dzięki bibliotece JCuda [102]. Uproszczona koncepcja algorytmu EMAS została przedstawiona na rysunku 4.1, gdzie ja widać EMAS jest głównym algorytmem generującym osobniki, a te następnie są optymalizowane za pomocą algorytmów SDLS oraz TABU przy użyciu procesora graficznego. Zarówno Konfiguracja platformy jak i koncepcja EMAS nie są tematem niniejszej rozprawy, więc nie będą te kwestie poruszane, a więcej na ich temat można znaleźć w [83]. Jednak, aby pokazać skuteczność opisanych algorytmów, posłużono się wykresem 4.8, który zawiera liczbę kroków EMAS wraz z współczynnikiem *metric factor* uzyskanym dla najlepszego uzyskanego rozwiązania, dzięki algorytmowi SDLS dla $L = 50$ (rysunek 4.8.a, 4.8.b) oraz $L = 64$ (rysunek 4.8.c, 4.8.d) . Dodatkowo liczbowe wartości dla tych przypadków zawarto w tabeli 4.3. Widać, że dzięki zastosowaniu układów GPGPU (w tym przypadku jest to karata NVIDIA GeForce GTX 1060 6GB) możliwe było przeszukanie w tym samym czasie (600 sekund) większej ilości rozwiązań w porównaniu z wersją, gdzie algorytmy heurystyczne zostały uruchomione w procesorze (w tym przypadku jest to procesor Intel Core i5-6600 3.30 GHz), przez co algorytm był wstanie znaleźć lepsze wyniki czyli te z wyższą wartością parametru metric factor. Warty podkreślenia jest fakt, że przy użyciu kart graficznych, większe przyśpieszenie uzyskano dla $L = 64$, gdzie wyniosło ono 4 razy, niż dla $L = 50$, gdzie wyniosło ono 2 razy. Należy zauważyć, że przy porównaniu wyników czasowych wykonanych tylko dla algorytmu SDLS zaimplementowanego na karcie graficznej z jego wersją uruchomioną w procesorze (rozdział 4.4), tendencja była odwrotna tzn. większe przyśpieszenia dla procesora graficznego zostały uzyskane dla krótszych ciągów. Jest to spowodowane faktem, że pomiar wykonany tylko dla algorytmu SDLS, dotyczył ciągów o długości $L = 48$ oraz $L = 201$ (tabele 4.1, 4.2), co jest sporą różnicą. Dokonując obliczeń dla dłuższego ciągu (201) istnieje potrzeba użycia przez wątek większej liczby rejestrów. W przypadku, gdy wymagana ilość tej pamięci jest większa niż ta dostępna na karcie, układ GPGPU zaczyna korzystać z pamięci globalnej, co powoduje spore straty czasowe. Konsekwencją tego faktu jest uzyskanie większego stosunku przyśpieszenia $\frac{gpgpu}{cpu}$, dla krótszych ciągów. Wyniki zaprezentowane w tym rozdziale dotyczą poszukiwania optymalnych rozwiązań dla ciągów o długości $L = 50$ oraz $L = 64$. Z architektury karty wynika, że dla obu tych długości, w każdym z bloków zostaną utworzone dwa warpy wątków (co zostało opisane w rozdziale 3.2), więc ilość niezbędnej pamięci oraz czas potrzebny do realizacji zadania na karcie graficznej, dla tych długości, będą bardzo zbliżone. W przypadku procesora obie te długości stanowią znaczącą różnicę, stąd też stosunek przyśpieszenia karta-procesor, dla ciągu o długości $L = 64$, jest większy. Dodatkowo należy wziąć pod uwagę fakt, że stosunek przyśpieszeń w przypadku EMAS został zmierzony dla całego algorytmu, czyli uwzględnia on oprócz optymalizacji lokalnej proces generowania osobników, których ilość może być w każdej iteracji inna (w rozdziale 4.4 liczba osobników była stała) co również może mieć wpływ na zaprezentowany stosunek przyśpieszeń.

Algorytm tabu rozwiązujący problem LABS, został uruchomiony dla długości $L = 64$ (rysunek

Tabela 4.3. Rezultaty dla koncepci EMAS z algorytmem SDLS dla wersji uruchomionej w procesorze oraz w karcie graficznej po 600 sekundach

Długość ciągu	CPU		GPGPU		Przyśpieszenie
	L. Kroków	Najlepsza wartość	L. Kroków	Najlepsza wartość	
50	2 000 000	8.17	4 000 000	8.17	2x
64	1 000 000	7.45	4 000 000	7.53	4x

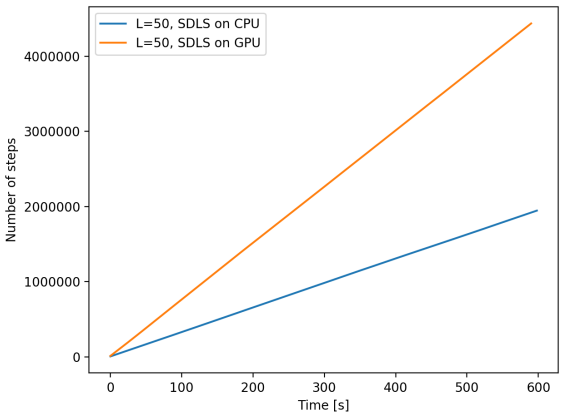
4.9a, 4.9b) oraz $L = 128$ (rysunek 4.9c, 4.9d) (dla długości krótszych, optima zostały znalezione dla obu wersji w bardzo krótkim czasie, co nie odzwierciedla korzyści użycia kart graficznych). W tym przypadku uzyskane przyśpieszenia, względem CPU wynoszą 25 dla $L = 64$ oraz 46 dla długości $L = 128$ razy. Oczywiste jest, że szybsze wersje potrzebowały znacznie mniej czasu na znalezienie optymalnych rozwiązań - wersja uruchomiona w CPU wcale nie znalazła optymalnego rozwiązania.
Warto podkreślić, że dla długości $L = 64$, która była użyta dla obu algorytmów, więc może służyć jako ich bezpośrednie porównanie, najlepsze rezultaty zostały uzyskane dla algorytmu tabu, dzięki któremu możliwe jest wyjście algorytmu z minimum lokalnego.

Tabela 4.4. Rezultaty dla koncepcji EMAS z algorytmem SDLS z tabu dla wersji uruchomionej w procesorze oraz w karcie graficznej po 600 sekundach

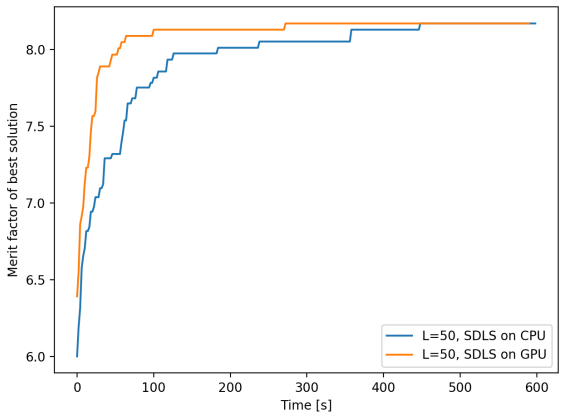
Długość ciągu	CPU		GPGPU		Przyśpieszenie
	L. kroków	Najlepsza wartość	L. kroków	Najlepsza wartość	
64	100 000	7.29	2 500 000	8.55	25x
128	30 000	5.69	800 000	6.06	46x

4.7. Propozycja nowego algorytmu SDLS-2 z lokalnością 2

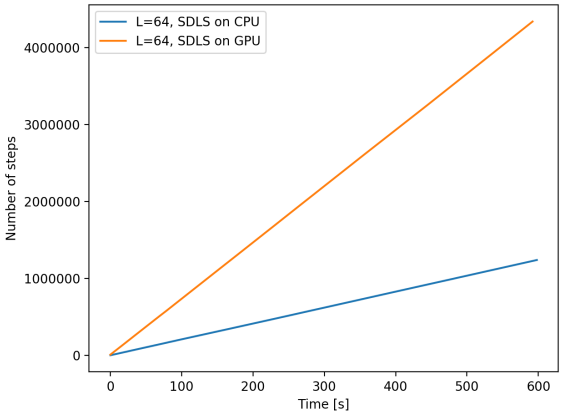
Nowe podejście do rozwiązania problemu LABS za pomocą innowacyjnego algorytmu nazwanego SDLS-2 (Alg. 7), polega na zwiększeniu obszaru przeszukiwań podczas jednej iteracji w taki sposób, by prócz poszukiwania najlepszego rozwiązania w sąsiedztwie oddalonym o jeden względem wejściowego ciągu (Alg. 7, linia 5), przeszukać również w obszarze oddalonym od niego o dwa bity (Alg. 7, linia 6). Dla sekwencji o długości L, rozwiązanie to implikuje konieczność przeszukania o $\frac{L(L-1)}{2}$ więcej rozwiązań niż w przypadku algorytmu SDLS z lokalnością jeden, co w ostateczności w pojedynczej iteracji wymaga przeszukania $\frac{L(L+1)}{2}$ ciągów, po czym podobnie jak dla zwykłego SDLS dla każdego z nich wyliczana jest energia. Mając obliczone enregie po zmianie jednego oraz dwóch bitów względem wejściowego ciągu, zostaje wyłoniona najlepsza z nich (Alg. 7, linia 7), która w przypadku, gdy jej wartość jest mniejsza od obecnej energii referencyjnej (Alg. 7, linia 8), staje się energią referencyjną do następnej iteracji (Alg. 7, linia 9), a ciąg z którego została wybrana (tym razem może różnić się również o 2 bity od wejściowego) zostaje użyty jako ciąg wejściowy do kolejnej iteracji (Alg. 7, linia 10). W sytuacji, gdy po przeszukaniu ciągów oddalonych o jeden oraz dwa bity od wejściowego, najlepsza



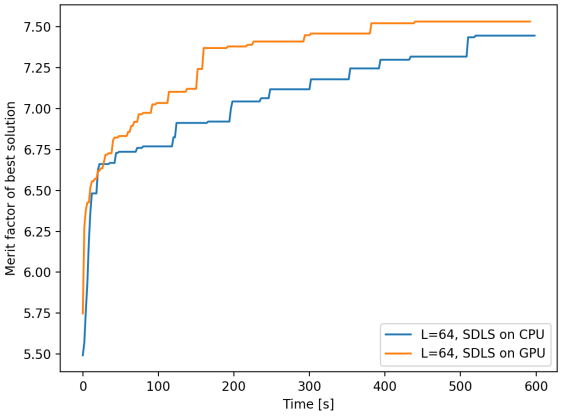
a. L. kroków algorytmu EMAS dla LABS 50



b. Współczynnik merit factor dla LABS 50



c. L. kroków algorytmu EMAS dla LABS 64

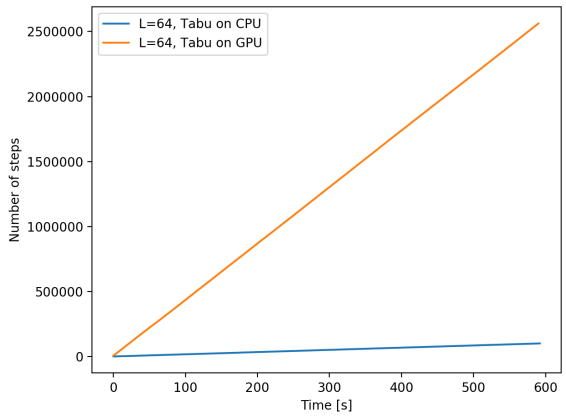


d. Współczynnik merit factor dla LABS 64

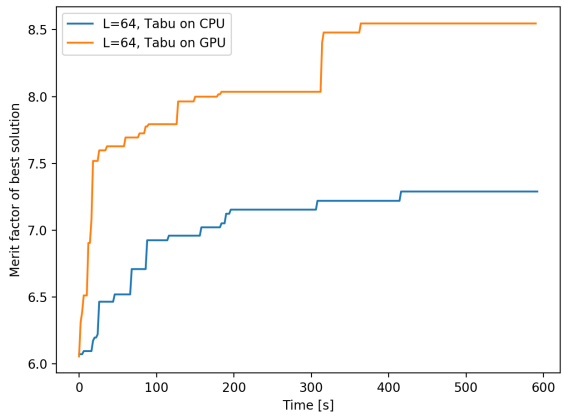
Rysunek 4.8. SDLS Wyniki dla LABS $L = 50$ (a, b) oraz $L = 64$ (c, d)

uzyskana w ten sposób energia nie jest lepsza od aktualnej energii referencyjnej, następuje przerwanie algorytmu, a obecna energia referencyjna E_r, wraz z ciągiem dla którego została obliczona, zostaje zwrócona jako optymalna energia, którą udało znaleźć się za pomocą opinanego algorytmu. Przykładowa iteracja dla sekwencji o długości $L = 5$ została pokazana na rysunku 4.10, gdzie po drugim obiegu kończy się algorytm, gdyż znaleziona w tym kroku energia nie jest większa od tej wyszukanej w kroku pierwszym.

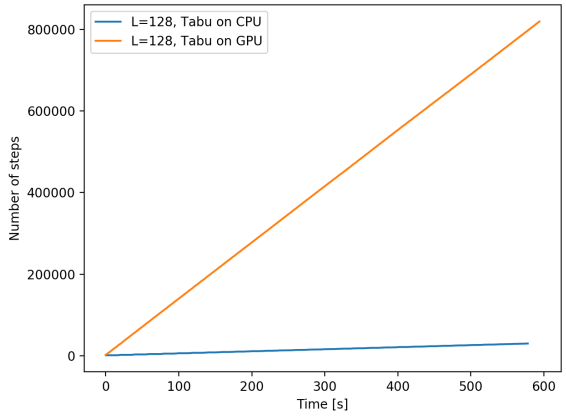
Implementacja w układach GPGPU omawianego podejścia używa dwóch pętli *zewnętrznej* oraz *wewnętrznej*. Rozwiązanie to wykorzystuje wiele funkcjonalności wywodzących się z wersji SDLS z lokalnością jeden. Podobnie jak tam, na początku budowane są struktury $C(S)$ oraz $T(S)$ (rysunek 4.2) (Alg. 8, linia 4), po czym na ich podstawie obliczana jest energia dla wejściowego ciągu E_r, stanowiąca energię referencyjną. Kolejnym krokiem jest obliczenie $L - 1$ energii, z których każda powstała w wyniku zmiany ciągu wejściowego o jeden. Zarówno obliczanie pomocniczych struktur jak i wartości energii po zmianie bitu oraz wybór najlepszej z nich, odbywa się w identyczny jak w implementacji dla zwykłego SDLS (użyte są te same funkcje), a których opis znajduje się w podrozdziale 4.4. Mając energie wyliczone z sąsiedztwa oddalonego o jeden bit, kolejnym krokiem algorytmu jest poszukiwanie rozwiązań w sąsiedztwie oddalonym dokładnie o dwa bity od sekwencji wejściowej. Aby tego dokonać



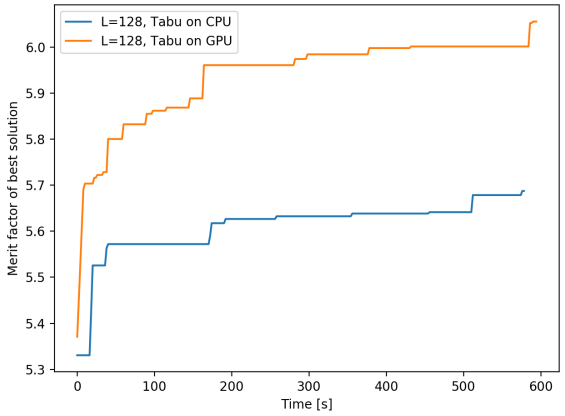
a. L. kroków algorytmu EMAS dla LABS 64



b. Współczynnik merit factor dla LABS 64



c. L. kroków dla algorytmu EMAS dla LABS 128



d. Współczynnik merit factor dla LABS 128

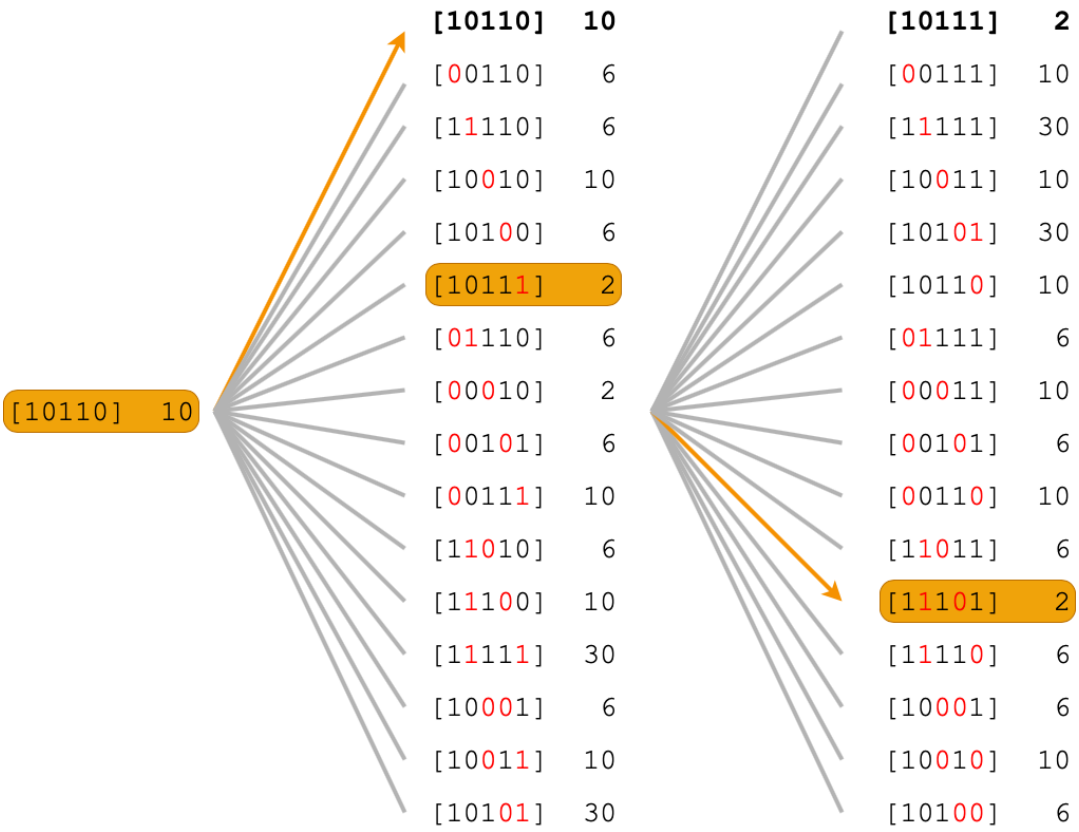
Rysunek 4.9. Tabu search Wyniki dla LABS $L = 64$ (a, b) and $L = 128$ (c, d)

należy użyć pętli, której liczba iteracji jest równa długości sekwencji $L-1$ (Alg. 8, linia 10). Pierwszym krokiem iteracji jest zmiana w wejściowej sekwencji bitu na pozycji równej aktualnemu obiegowi pętli, co dokonuje się przez wątek o takim indeksie (Alg. 8, linia 11). Mając sekwencję ze zmienionym bitem, która staje się podstawową do dalszych obliczeń, należy zaktualizować struktury pomocnicze $C(S)$ oraz $T(S)$ (Alg. 8, linia 12). Wspomniana aktualizacja dokonywana jest w taki sam sposób dla dla algorytmu SDLS z lokalnością jeden, gdy zostaje znaleziona lepsza energia (rozdział 4.4). Następnie dla ciągu, który różni się na jednej pozycji od ciągu wejściowego, obliczanych jest $L-1$ energii, przy użyciu $L-1$ wątków w taki sam sposób jak było to zrobione dla lokalności jeden (Alg. 8, linie 13 -15)). Dzięki tym dwóm krokom, otrzymanych zostaje $L-1$ energii, powstałych z sekwencji różniących się o dwa bity od sekwencji wejściowej. Powodem, dla którego w pierwszej iteracji obliczanych jest $L-1$ energii przy użyciu $L-1$ wątków zamiast obliczać L energii przy użyciu L wątków jest fakt, że w pierwszym kroku algorytmu zostały obliczone wartości energii dla ciągów różniących się o jeden bit od sekwencji wejściowej, więc przy założeniu że dla przykładowego wejściowego ciągu o długości $L = 5$ składającego się z samych jedynek, zgodnie z algorytmem wejściem do pierwszej iteracji, jest ciąg $-1, 1, 1, 1, 1$. Gdyby wątek o indeksie zero brał udział w kalkulacjach, to w iteracji zero ciąg dla którego obliczałby on energię, byłby w postaci również samych jedynek, a taka energia została już policzona przy okazji SDLS

Algorytm 7. Sekwencyjna wersja algorytmu SDLS-2

```
1: function SEQUNTIALSDLS-2(S)
2:     improvement := true
3:     E_r = compute_reference_energy(S)
4:     while improvement do
5:         E_local_I = calcuate_energies_by_mutation_of_single_bit(S)
6:         E_local_II = calcuate_energies_by_mutation_of_two_bits(S)
7:         E_best=compute_lowest_energy(E_local_I, E_local_II)
8:         if E_r < E_best then improvement := false
9:         update_reference_energy()
10:        update_reference_sequence()
       end while
11:    return E_best
   end function
```

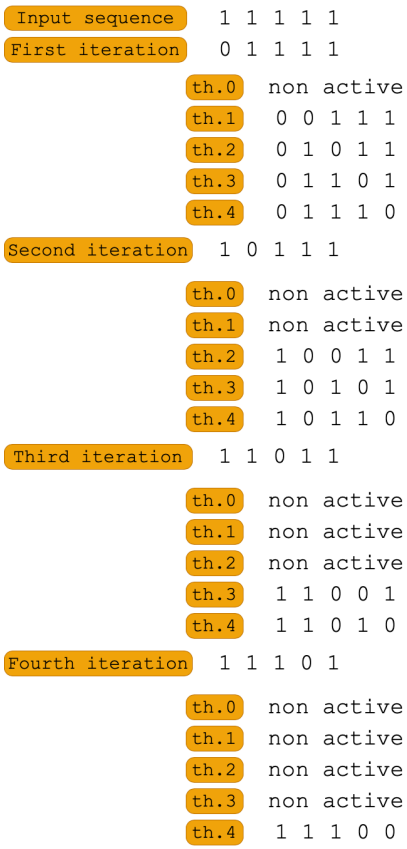
z lokalnością jeden (przy okazji liczenia pierwszej energii), więc nie ma sensu jej obliczać drugi raz. Z tak wyliczonych $L-1$ energii wybierana i zapisywana jest najlepsza z nich. Podobnie jak poprzednio, wyboru dokonuje się używając operacji podwójnej redukcji, zwracając wraz z najniższą energią wartość pozycji, na której się znajdowała. Jest to koniecznie by móc otworzyć ciąg, dzięki któremu została ona obliczona. W iteracji numer dwa, ciąg wejściowy zostaje zmieniony na pozycji numer jeden, czego dokonuje wątek o takim indeksie, po czym zostaje on wyłączony z kalkulacji. W tej iteracji również wątek o indeksie zero jest nieaktywny, gdyż jego obliczenia w zaproponowanym algorytmie doprowadziłyby do obliczenia ciągu w postaci, dla której energia była już obliczona, gdyż dla wyżej przedstawionego ciągu wejściowego dla drugiej iteracji wejściem jest ciąg : $1, -1, 1, 1, 1$, gdyby ciąg zero brał udział w obliczeniach jego ciągiem byłby w postaci $-1, -1, 1, 1, 1$, a taki był użyty w iteracji pierwszej dla wątku z indeksem jeden. Natomiast w bieżącej iteracji, ciągiem dla którego obliczałby energię wątek z tym indeksem, byłby ciąg $1, 1, 1, 1, 1$, a jest to dokładnie ciąg wejściowy do całego algorytmu, dla którego energia była policzona jako pierwsza. W związku z tym w drugiej iteracji wyliczanych jest $L-2$ energii, z których podobnie jak poprzednio wybierana jest najlepsza wraz z wątkiem przez który została obliczona. Energia ta jest porównywana z energią uzyskaną z pierwszego kroku i mniejsza z nich jest ustawiana jako obecna najlepsza energia uzyskana po zmianie dwóch bitów w ciągu wejściowym (Alg. 8, linia 16). W tym momencie, aby możliwe było późniejsze odtworzenie ciągu dla którego została obliczona finalna najlepsza energia, należy zapamiętać informację, w której iteracji została ona znaleziona (gdyby w drugiej iteracji energia pochodząca z niej była mniejsza od pierwszej wtedy iteracja jeden zostaje zapisana). Mając indeks, dzięki któremu została znaleziona najlepsza energia oraz iteracja, w której to się dokonało, możliwe jest z ciągu wejściowego odtworzyć ciąg, dla którego optymalna energia została znaleziona. Dodatkowo, w tym kroku należy aktualizować pomocniczą strukturę $C(S)\_optimum$, wartościami pochodzącymi ze zwycięskiego wątku. Struktura ta jest nową konieczną strukturą, gdyż na tym etapie algorytmu nie wiadomo, czy najlepsza energia zostanie wyłoniona spośród rozwiązań algorytmu SDLS z lokalnością jeden czy jego wersją z lokalnością dwa. Struktura ta po obliczeniu wartości energii po zmianie jednego bitu zawierała wartości pochodzące z pamięci podręcznej zwycięskiego wątku.



Rysunek 4.10. Przykład iteracji w algorytmie SDLS z lokalnością 2

W momencie wyłonienia najlepszej energii po zmianie dwóch bitów w danej iteracji należy sprawdzić czy otrzymana w ten sposób energia jest mniejsza od tej najlepszej uzyskanej po zmianie jednego bitu. Jeśli tak, to należy dokonać aktualizacji struktury $C(S)\_optimum$ (Alg. 8, linia 17), używając wartości pochodzących z pamięci podręcznej zwycięskiego wątku w danej iteracji, algorytmu poszukującego rozwiązań z lokalnością dwa.

W kolejnych iteracjach uzyskiwanych jest kolejno o jedno mniej rozwiązanie przy użyciu o jednego mniej wątku. Reszta kroków jest powtarzana. Sytuację tę, dla $L = 5$ obrazuje rysunek 4.11. Mając wyliczone wszystkie energie, po przeszukaniu przestrzeni w sąsiedztwie oddalonym o jeden oraz dwa, wyłaniana jest ta najlepsza, po czym odtwarzany jest ciąg, dla którego została ona obliczona. Następnie należy sprawdzić warunek czy otrzymana energia jest lepsza od obecnej referencyjnej (w przypadku pierwszej iteracji referencyjną energią jest ta obliczona dla ciągu wejściowego), jeśli tak to następuje podmiana energii referencyjnej, ciągu referencyjnego oraz aktualizacja struktur $C(S)$ oraz $T(S)$ (Alg. 8, linie 19- 21). W przypadku, gdy najlepsza energia zwrócona z SDLS_2, jest mniejsza od obecnej referencyjnej, wówczas algorytm jest przerywany, gdyż w algorytmie SDLS kolejne iteracje powodowałyby zwracanie dokładnie takiej samej wartości, gdyż w momencie braku poprawy algorytm nie podmienia sekwencji, ponieważ z definicji wejściową sekwencją jest zawsze ta najlepsza. Opisany algorytm dotyczy pojedynczego bloku. Podobnie jak w przypadku SDLS z lokalnością jeden, takich bloków jest obliczanych K naraz, więc na wyjściu algorytmu zostaje zwrócone K energii, z których przy użyciu algorytmu redukcji (zwracającej wartość najlepszej energii wraz z indeksem pod, którym się znajdowała), zostaje



Rysunek 4.11. Obliczanie energii przez wątki w algorytmie SDLS-2

wyszukana najlepsza. Jak zostało wspomniane algorytm w obrębie jednego bloku działa tak długo, aż będzie uzyskane polepszenie względem poprzedniego kroku. Jednak należy mieć na uwadze, że każdy blok otrzymuje unikatowy ciąg wejściowy (w miarę możliwości), więc każdy może skończyć poszukiwania energii w innym momencie czasowym. W związku z tym czas wyłaniania globalnej najlepszej energii jest równy czasowi, w jakim poszukiwania realizował najdłużej poszukujący blok wątków.

4.8. Propozycja algorytmu SDLS z przeszukiwaniem w głąb

Drugą zaproponowaną modyfikacją algorytmu SDLS rozwiązującą problem LABS jest algorytm nazwany *SDLS przeszukujący w głąb* (Alg. 9). Algorytm podobnie jak SDLS-2 oparty jest na dwóch pętlach *zewnętrznej* i *wewnętrznej*. Liczba generowanych przez algorytm rozwiązań w pojedynczym kroku jest nie możliwa do obliczenia, gdyż wewnętrzna pętla algorytmu kończy działanie w momencie braku poprawy (Alg. 9, linia 9), a nie jak poprzednio po przeszukaniu wszystkich rozwiązań oddalonych o jedn oraz o dwa względem wejściowego ciągu. W tym przypadku obliczanie rozwiązań w sąsiedztwie jeden oraz dwa odbywa się w jednym kroku, a nie jak poprzednio w dwóch (w SDLS-2 na początku liczone były energie po zmianie o jeden, następnie po zmianie o dwa i z nich wybierana była najlepsza). Przez jeden krok algorytmu należy rozumieć jeden obieg pętli zewnętrznej (Alg. 9, linia 3), w której ciele, jako pierwsza obliczana jest pojedyncza energia po zmianie jednego bitu na pozycji równej obiegowi pętli (Alg. 9, linia 4), po czym obliczanych jest L energii, powstałych w wyniku zmiany dwóch bitów

Algorytm 8. Równoległa wersja SDLS-2 w GPGPU

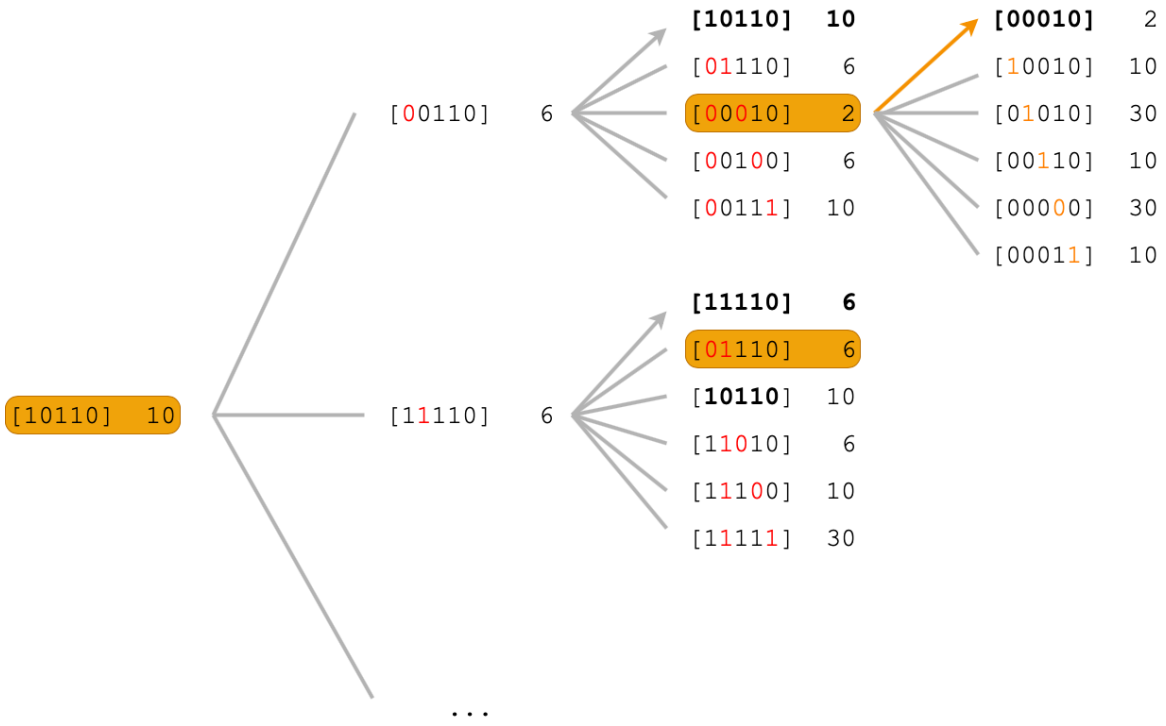
```
1:  function PARALLELSDLS(S)
2:      transfer_solutions_to_GPU_blocks(S)
3:      for block := 0 to len(S) do
4:          create_T(S)_and_C(S)()
5:          E_r = compute_reference_energy(S_block)
6:          while improvement do
7:              mutation_of_threadId_bit(S_block)
8:              ParallelValueFlip(S_block, T', C')
9:              E_best := E_best_local1 := compute_lowest_energy()
10:             for bit := 0 to len(L − 1) do
11:                 if threadId == bit then S_block[threadId]* = −1 end if
12:                 update_T(S)_and_C(S)()
13:                 mutation_of_threadId_bit(S_block)
14:                 ParallelValueFlip(S_block, T', C')
15:                 E_rlocal2 := compute_lowest_energy_for_current_bit()
16:                 if E_r_local2[bit] < E_r_local2[bit − 1] then E_best_local2 := E_r_local2[bit] end if
17:                 if E_best_local2 < E_best_local1 then update_C(S)_optimum_and_E_best end if
                end for
18:             if E_r < E_best then improvement := false
19:             update_reference_energy()
20:             update_reference_sequence()
21:             update_T(S)_and_C(S)()
            end while
        end for
22:     return E_best
    end function
```

względem wejściowego ciągu (Alg. 9, linia 7). Z tak uzyskanych energii wybierana jest najlepsza (Alg. 9, linia 8), i w momencie, gdy jej wartość jest mniejsza od obecnej energii referencyjnej (Alg. 9, linia 9), po aktualizacji E_r (Alg. 9, linia 10) oraz odpowiadającej jej sekwencji (Alg. 9, linia 11), z takimi wartościami rozpoczyna się kolejny obieg pętli wewnętrznej. W kolejnych obiegach pętli wewnętrznej nie ma możliwości stwierdzenia o ile bitów przeszukiwane rozwiązania różnią się od ciągu wejściowego, gdyż jest to zależne od zwycięskiej sekwencji, która może różnić się o jeden bądź dwa bity względem wejściowej. Końcem pojedynczego kroku jest koniec pętli *while*. Po każdym zakończeniu kroku następuje w miarę potrzeby aktualizacja obecnie najlepszego wyniku, po czym rozpoczyna się kolejny krok. Całkowita liczba takich kroków algorytmu jest równa długości przeszukiwanego ciągu. Na rysunku 4.12 pokazano dwa pierwsze kroki (dwie iteracje zewnętrzne) omawianego algorytmu. Widać że w tym przypadku kroki kończą się po różnej liczbie iteracji pętli wewnętrznej (w 1 roku to 2 iteracje, w drugim jedna), gdyż jak zostało to wcześniej powiedziane ta kończy swoje poszukiwania w momencie braku poprawy.

Algorytm 9. Sekwencyjna wersja algorytmu SDLS deep into

```
1:  function SEQUNTIALSDLS-DEEPINTO(S)
2:      E_r = compute_reference_energy(S)
3:      for i := 0 to len(L) do
4:          E[i] = compute_single_energy_by_mutation_i_th_bit(S)
5:          improvement := true
6:          while improvement do
7:              E_local_II = calcuate_energies_by_mutation_of_two_bits(S)
8:              E_best=compute_lowest_energy(E[i], E_local_II)
9:              if E_r < E_best then improvement := false
10:             update_reference_energy()
11:             update_reference_sequence()
            end while
        end for
12:     return E_best_global
    end function
```

W implementacji w GPGPU, pierwszym krokiem algorytmu jest stworzenie zewnętrznej pętli o długości L (Alg. 10, linia 4). Podobnie jak w $SDLS-2$, pierwszym krokiem tej pętli jest zmiana bitu w ciągu wejściowym na pozycji równej licznikowi pętli, oznaczanemu jako bit, co jest czynione poprzez wątek o takim indeksie (Alg. 10, linia 4). Dla zmienionego w ten sposób ciągu wejściowego obliczane są struktury $C(S)$ oraz $T(S)$ (rysunek 4.2) (Alg. 10, linia 6). Następnie z ciągu różniącego się o jeden bit od ciągu wejściowego, obliczana jest pierwsza energia, co dostarcza pierwszego rozwiązania dla sąsiedztwa jeden, oraz staje się ona energią referencyjną dla drugiego kroku algorytmu (Alg. 10, linia 7). Ciąg różniący się na jednej pozycji, staje się zarazem wejściem do drugiego kroku algorytmu, którym jest poszukiwanie rozwiązania *w głąb*. Istotą przeszukiwania *w głąb* jest szukanie rozwiązania tak długo, aż kolejne rozwiązania nie będą przynosić poprawy, co zrealizowane jest poprzez pętle wewnętrzną *while* (Alg. 10, linia 8), która w SDLS-2 była pętlą zewnętrzną. W zaproponowanym podejściu w każdej iteracji pętli wewnętrznej obliczanych jest L energii, powstałych po zmianie jednego bitu, tym razem względem ciągu różniącego się o jeden od ciągu wejściowego do pętli (różniącego się na jednej pozycji względem wejściowego) (Alg. 10, linie 9-10). Obliczenia odbywają się według algorytmu SDLS (rozdział 4.4). W tym przypadku inaczej jak w SDLS-2, nie jest blokowany wątek o indeksie równym indeksowi pętli, gdyż zgodnie z algorytmem obliczy on energię dla oryginalnego ciągu wejściowego (w SDLS-2 była to energia referencyjna). Z L obliczonych energii wybierana jest najlepsza E_{best} i porównywana jest z energią obliczoną po zmianie jednego bitu E_r, na pozycji bit czyli równej licznikowi pętli zewnętrznej. Jeśli energia E_{best} jest lepsza od E_r (Alg. 10, linia 12), wówczas zwycięski wątek aktualizuje struktury $C(S)$ oraz $T(S)$, energia E_{best} staje się energią referencyjną, a sekwencja dzięki której ją obliczoną sekwencją wejściową do drugiego kroku algorytmu, czyli drugiego obiegu pętli *while* (Alg. 10, linie 13-15). Należy zauważyć, że przeszukiwania mogą odbywać się w odległości dużo większej niż dwa od wejściowego ciągu, gdyż wejściem do drugiego obiegu może być ciąg różniący się na dwóch bitach od ciągu wejściowego, lub może to być ciąg identyczny z ciągiem wejściowym, co będzie miało



Rysunek 4.12. Przykład iteracji w algorytmie SDLS z przeszukiwaniem w głąb

miejsce, gdy najlepsza energia zostanie obliczona dla ciągu z indeksem równym bit. W pierwszym przypadku mając ciąg różniący się o dwa bity, w drugim obiegu pętli *while* nastąpi przeszukanie oddalone o dwa bity od swojego wejścia, co jest równoznaczne z oddaleniem o cztery bity od oryginalnego ciągu wejściowego. Odległość między oryginalnym ciągiem wejściowym, a aktualnie przeszukiwanym w pętli *while*, zwiększa się wraz z każdym obiegiem, stąd też w tym przypadku w kolejnych obiegach wątki nie są blokowane, gdyż stwierdzenie, które sekwencje były już przeliczone nie jest w tym przypadku możliwe. W momencie wystąpienia *break* w pętli *while*, rozpoczyna się kolejny obieg pętli zewnętrznej, czyli tej iterującej po kolejnych bitach ciągu wejściowego. W przypadku drugiego obiegu, zostaje zamieniony bit w ciągu wejściowym znajdujący się na drugiej pozycji, co daje drugą energię obliczoną po zmianie dokładnie jednego bitu w stosunku do ciągu wejściowego i ten ciąg wraz z jego energią, dalej stają się wejściami do drugiego kroku algorytmu czyli *przeszukiwaniu w głąb*. W każdym obiegu pętli zewnętrznej, w razie potrzeby aktualizowana jest, wartość najlepszej uzyskanej dotychczas energii $E_{best\,global}$ przy użyciu wartości E_{best} (Alg. 10, linia 16) (w przypadku, gdy pierwsza energia po zmianie jednego bitu ma niższą wartość niż ta obliczona po pierwszym obiegu *while*, wówczas to ona będzie obecną najlepszą energią globalną). Energia $E_{best\,global}$ jest wartością zwracaną przez algorytm. Aby lepiej przybliżyć pracę wątków, należy rozważyć przetwarzane przez nie wartości dla pierwszego obiegu, zarówno zewnętrznej jak i wewnętrznej pętli. Podobnie jak poprzednio niech ciąg wejściowy o długości $L = 5$ będzie wypełniony samymi jedynkami. Wówczas w pierwszej iteracji po zmianie bitu na pozycji zero otrzymany zostanie ciąg $-1, 1, 1, 1, 1$. Z tego ciągu obliczana jest pierwsza energia, będąca

pierwszym rozwiązaniem z puli rozwiązań, powstałych z ciągów różniących się na jednej pozycji od wejściowego (pierwsza energia dla SDLS z lokalnością jeden). Następnie każdy wątek zmienia jeden bit i zgodnie z funkcją SDLS z rozdziału 4.4 oblicza swoją energię. Ponieważ wątek z indeksem zero nie jest tym razem zablokowany, obliczy on energię dla ciągu z wartościami: $1, 1, 1, 1, 1$, co daje wartość energii ciągu wejściowego, która w poprzednich podejściach była obliczana jako pierwsza. W tym momencie jest ona zwracana przez wątek, którego indeks odpowiada obiegowi pętli zewnętrznej podczas pierwszej iteracji pętli wewnętrznej. W kolejnych obiegach pętli *while*, wątki otrzymują nowe ciągi referencyjne i postępują zgodnie z algorytmem SDLS. Kolejne obiegi pętli zewnętrznej odbywają się w analogiczny sposób, a jedyna zmiana dotyczy bitu, na którym zostaje zmieniony oryginalny ciąg, który musi być przechowywany w osobnej strukturze, tak by możliwe było jego użycie w kolejnych iteracjach pętli zewnętrznej.

Algorytm 10. Równoległa wersja SDLS z przeszukiwaniem w głąb w GPGPU

```
1:  function PARALLELSDLS(S)
2:      transfer_solutions_to_GPU_blocks(S)
3:      for block := 0 to len(S) do
4:          for bit := 0 to len(L) do
5:              if threadId == bit then S_block[threadId]* = −1 end if
6:              create_T(S)_and_C(S)()
7:              E_r := compute_energy_with_local_one(S_block)
8:              while improvement do
9:                  mutation_of_threadId_bit(S_block)
10:                 ParallelValueFlip(S_block, T', C')
11:                 E_best := compute_lowest_energy()
12:                 if E_r < E_best then improvement := false end if
13:                 update_reference_energy()
14:                 update_reference_sequence()
15:                 update_T(S)_and_C(S)()
                end while
16:             if E_bestglobal < E_best then E_best_global := E_best end if
            end for
        end for
17:     return E_best_global
    end function
```

Wyżej opisane przeszukiwanie, dotyczy jednego bloku. Podobnie jak we wszystkich opisanych do tej pory algorytmach rozwiązujących problem LABS, tutaj też uruchomionych jest *K* takich bloków, z których każdy otrzymuje możliwe unikalną sekwencję wejściową. W momencie zakończenia obliczeń przez ostatni blok, uruchamiana jest redukcja poszukująca, najniższego rezultatu z bloków wraz z indeksem tego zwycięskiego (by możliwe było wyszukanie zwycięskiej sekwencji).

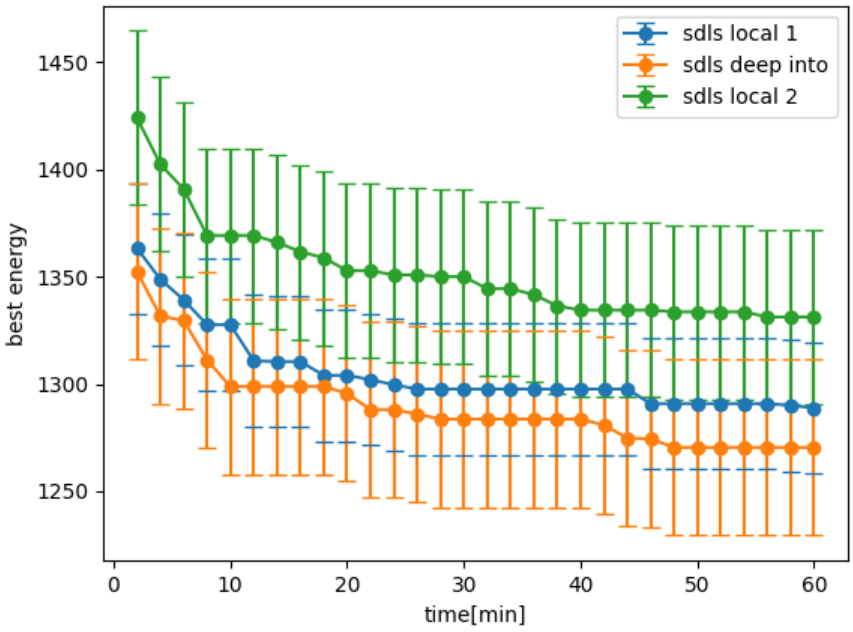
Tabela 4.5. Porównanie ilości przeszukanych osobników dla L=128 przez trzy algorytmy SDLS w czasie jednej minuty

Metoda	Liczba przeszukanych rozwiązań	Średnia	Odchylenie standardowe	Liczba uruchomionych kerneli
SDLS	20 740 434 073	2807	531.8	57773
SDLS-2	16 865 830 144	224 089	41837.6	593
SDLS W GŁĄB	31 176 794 592	356 752	46302.4	703

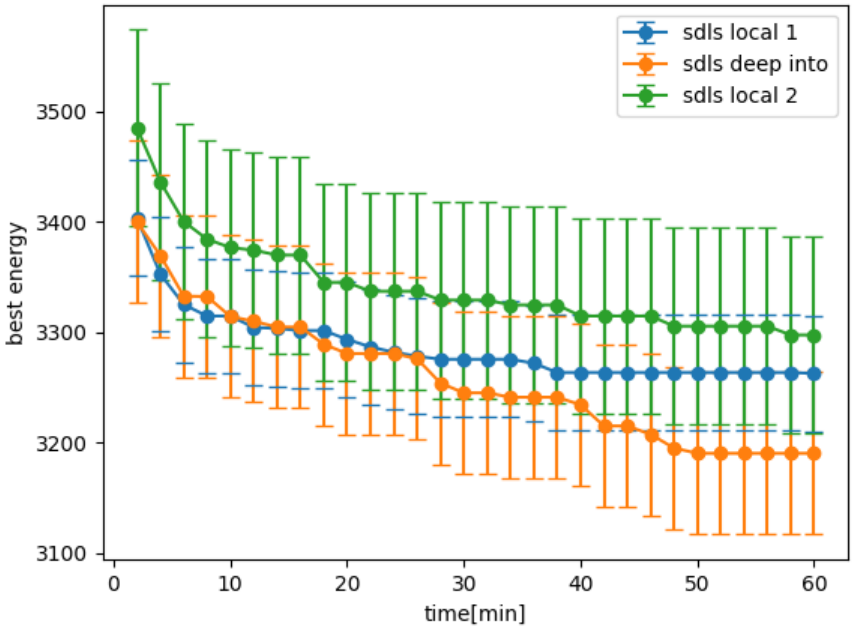
4.9. Pomiar skuteczności algorytmów SDLS z lokalnością 2 oraz SDLS z przeszukiwaniem w głąb dla problemu LABS

W celu sprawdzenia skuteczności algorytmów SDLS-2 oraz SDLS z przeszukiwaniem w głąb, w porównaniu z wersją SDLS z lokalnością jeden, każdy z nich poszukiwał optymalnego rozwiązania dla trzech długości wejściowych sekwencji $(128, 192, 256)$ przez okres jednej godziny. W pierwszym obiegu za pomocą procesora zostaje wylosowanych 128 sekwencji, po jednej dla każdego z bloków. Z tak wygenerowanymi danymi wejściowymi, odpowiedni kernel rozpoczyna poszukiwania minimalnej energii. W momencie, gdy z wszystkich bloków zostaje wyłoniona najlepsza energia (czyli w żadnym bloku w kolejnej iteracji algorytm nie był wstanie znaleźć lepszego rozwiązania), jest ona zapamiętywana jako obecnie najlepsza energia globalna $E_{global\_optimum}$. Po czym na procesorze następuje nowe losowanie ciągów wejściowych, z którymi algorytm rozpoczyna kolejne poszukiwania, a najniższa energia jaką zdołał uzyskać tym razem, jest porównywa z dotychczasową globalną minimalną energią $E_{global\_optimum}$ i w razie polepszenia, obecne globalne optimum jest aktualizowane. Procedura jest powtarzana, aż warunek czasowy (1h) nie zostanie przekroczony. Dodatkowo co dwie minuty zapisywany jest aktualny najlepszy rezultat. Cała procedura dla każdego z trzech algorytmów została powtórzona 10 razy.

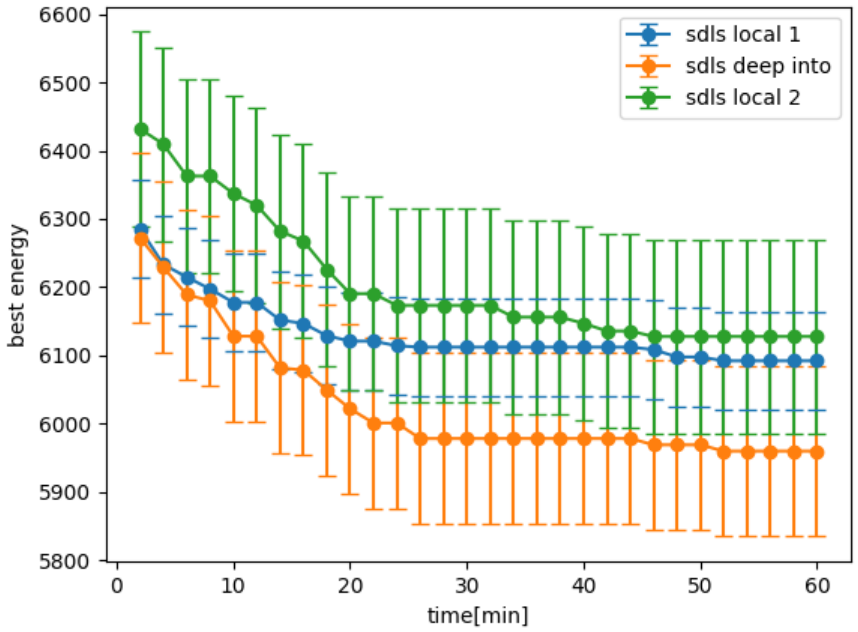
Tabele 4.5, 4.6 oraz 4.7 zawierają ilość wszystkich przeszukanych ciągów wraz ze średnią przeszukanych rozwiązań dla jednego bloku oraz liczbę uruchomionych osobnych przeszukiwań (ilość wywołań funkcji w GPGPU) dla trzech zaproponowanych algorytmów. Skuteczność algorytmów została zaprezentowana na wykresach 4.13, 4.14 oraz 4.15. Widać, że najbardziej skutecznym algorytmem dla wszystkich długości ciągów okazał się algorytm *SDLS z przeszukiwaniem w głąb*, gdyż jego implementacja w GPGPU jest najbardziej wydajna, ponieważ w wyznaczonych ramach czasowych (1h) jest wstanie przeszukać więcej osobników niż algorytmy SLDS i SLDS-2 kolejno o 50% i 84% dla $L = 128$, o 34% i 68% dla $L = 192$ oraz o 12% i 56% dla $L = 256$. Skuteczność algorytmu SDLS-2, choć najsłabsza, jest porównywalna z podstawową wersją algorytmu SDLS, mimo że w zaproponowanej implementacji na procesorze graficznym przeszukuje on najmniej z wszystkich osobników. Fakt ten świadczy, że sama skuteczność algotymu jest wysoka, jednak zaprezentowana implementacja jest mało wydajna. Poprawa efektywności implementacji w GPGPU stwarza pole do dalszych prac badawczych.



Rysunek 4.13. Zmiana energii w czasie dla L=128



Rysunek 4.14. Zmiana energii w czasie dla L=192



Rysunek 4.15. Zmiana energii w czasie dla L=256

Tabela 4.6. Porównanie ilości przeszukanych osobników dla L=192 przez trzy algorytmy SDLS w czasie jednej minuty

Metoda	Liczba przeszukanych rozwiązań	Średnia	Odchylenie standardowe	Liczba uruchomionych kerneli
SDLS	19 152 109 696	5458	1116	27303
SDLS-2	15 342 863 872	818216	145562	148
SDLS W GŁĄB	25 827 182 720	1056786	121272	191

Tabela 4.7. Porównanie ilości przeszukanych osobników dla L=256 przez trzy algorytmy SDLS w czasie jednej minuty

Metoda	Liczba przeszukanych rozwiązań	Średnia	Odchylenie standardowe	Liczba uruchomionych kerneli
SDLS	22 380 865 408	11159	2 836	15 661
SDLS-2	15 975 437 994	1 657 780	257 819	74
SDLS DEEP INTO	24 978 089 130	3 019 130	292 121	68

5. Trenowanie algorytmu wektorów nośnych ze zredukowaną precyzją danych w celu klasyfikacji tekstu

Rozdział ten poświęcony jest klasyfikacji tekstu, przy pomocy algorytmu wektorów nośnych. Szczególny nacisk został położony na zalety zastosowania procesu kwantyzacji oraz jej wpływu na skuteczność użytego algorytmu. Przetwarzanie tekstu należy do grupy algorytmów związanych z przetwarzaniem języka naturalnego, który jest istotną gałęzią sztucznej inteligencji. Z tego też względu w rozdziale tym zaprezentowano najczęściej stosowane rozwiązania w przetwarzaniu języka naturalnego, począwszy od tych najprostszych, po obecnie stosowane, które osiągają bardzo wysoką skuteczność w wielu zadaniach związanych z przetwarzaniem języka naturalnego. Jednocześnie algorytmy te wymagają olbrzymiej mocy obliczeniowej, a co za tym idzie, do ich wykonania musi zostać zużyta ogromna ilość energii. Jak zostało pokazane na przykładzie klasyfikacji tekstu przy pomocy algorytmu wektorów nośnych, użycie kwantyzacji redukuje czas potrzebny na dokonanie obliczeń, co prócz faktu, iż zjawisko to jest bardzo pożądane, szczególnie w odniesieniu do algorytmów które powinny działać w czasie rzeczywistym, to również finalnie prowadzi do oszczędności energii. Klasyfikacja tekstu bazująca na algorytmie z nadzorem jest stosunkowo prostym zadaniem, więc do reprezentacji tekstu w formie liczbowej został użyty algorytm TF-IDF (rozdział 5.1.2), który nie jest wymagający jeśli chodzi o moc obliczeniową. Główną intencją opisywanej implementacji było pokazanie wpływu użycia zredukowanej precyzji danych w procesie uczenia klasyfikatora, a później w samej klasyfikacji, na skuteczność algorytmu. Wiedza ta może zostać wykorzystana przy próbie kwantyzacji innych dużo bardziej złożonych modeli, dlatego też zostały one opisane w niniejszym rozdziale obok zaproponowanych metod kwantyzacji. Dodatkowo w rozdziale tym został szczegółowo opisany algorytm wektorów nośnych wraz z procesem jego uczenia, który poddawany jest kwantyzacji, co stanowi najważniejszą część niniejszego rozdziału.

5.1. Przetwarzanie języka naturalnego

Przetwarzanie języka naturalnego (ang. *Natural language processing - NLP*) obok przetwarzania obrazów i wideo jest uważane za jedną z trzech głównych gałęzi uczenia głębokiego, dostarczającej aplikacjom możliwości rozumienia i przetwarzania języka ludzkiego, reprezentowanej w formie mówionej bądź pisanej. Modele NLP są zdolne do:

– rozumienia języka naturalnego,

– tworzenia gramatycznych modeli zdań,

– automatycznego tłumaczenia z jednego języka na inny,

– generowaniu języka naturalnego,

– wyszukiwaniu tekstu według zawartości semantycznej,

– określenie podobieństwa dwóch dokumentów,

– klasyfikacji tekstu,

– kategoryzacji tekstu,

– automatycznej poprawy pisowni,

– tworzenia programów potrafiących w inteligenty sposób komunikować się z człowiekiem np. czatboty.

Do przetwarzania języka naturalnego można z powodzeniem zastosować ogólne techniki uczenia maszynowego bądź głębokiego uczenia maszynowego. Jednak w celu uzyskania wysokiej wydajności, podobnie jak w innych obszarach sztucznej inteligencji, ważne stają się strategie związane z daną dziedziną. Aby zbudować wydajny model NLP należy użyć technik wyspecjalizowanych w przetwarzaniu danych sekwencyjnych. Podstawowymi algorytmami używanymi do pracy z sekwencjami są **rekurencyjne sieci neuronowe** (rozdział 2.2.2), **jednowymiarowe konwolucyjne sieci neuronowe** (rozdział 2.2.1) czy jak w przypadku niniejszej pracy algorytm **wektorów wspierających**, którego dokładny opis znajduje się w dalszej części rozdziału. W naszych eksperymentach, odnośnie przetwarzania danych sekwencyjnych, skupiamy się na klasyfikacji. Modele uczenia głębokiego nie są przystosowane do przetwarzania tekstu w surowej formie tylko w jego reprezentacji numerycznej, która podawana jest na wejście modelu w postaci tensora. **Wektoryzację**, czyli mapowanie tekstu do modelu przestrzeni wektorowej (ang. *Vector Space Model - VSM*), rozpoczyna się od podziału tekstu na **tokeny**, czyli jednostki którymi mogą być znaki, pojedyncze wyrazy, zlepki kilku kolejnych wyrazów (n-gramy) lub jak w metodzie *Sentcepiece* [58] fragmenty słów tzw. *subwords*. Podział tekstu na tokeny nosi nazwę **tokenizacji**. Po zastosowaniu wybranego schematu tokenizacji, następuje zamiana tekstu na jego reprezentację numeryczną, która może być przeprowadzona na wiele sposób, z których najważniejsze zostały opisane w dalszej części rozdziału.

5.1.1. Model reprezentacji tekstu *Bag of words*

Najprostszym modelem wektorowym jest tzw. *Bag of Words (BoW)*. W modelu tym dokument jest reprezentowany jako zbiór słów, nie uwzględniający gramatyki ani kolejności ich wystąpienia. Metoda polega na definiowaniu unikatowych słów oraz zliczaniu ich wystąpień. Wektor zatem zawiera unikalny indeks słowa oraz przypisaną mu liczbę wystąpień. Reprezentację tę można porównać do histogramu. Wadą takiego podejścia jest bardzo duży wzrost wielkości wektora w momencie, gdy nowe dokumenty zawierają nowe słowa, tym bardziej że metoda nie jest odporna na odmianę gramatyczną ani na kolejność słów. W metodzie tej reprezentacja wektorowa często ma postać wektora rzadkiego, gdyż niektóre słowa mogą występować tylko raz, w związku z czym ich indeksowi jest przypisana wartość zero, co oznacza

pojedyncze wystąpienie. Największą wadą metody BoW jest nieumiejętność rozróżnienia znaczenia semantycznego, spowodowana ignorowaniem kontekstu wystąpienia słowa. W wyniku tego model nie jest zdolny do wskazania różnicy między zdaniem twierdzącym : *"this is interesing"* a zawierającym te same słowa zdaniem pytającym *"is this interesing"*.

5.1.2. Metoda *Term frequency - inverse document frequency - TF-IDF*

Jedną z najczęściej używanych metod służącą do numerycznej reprezentacji tekstu, bazującej na tokenach zbudowanych z pojedynczych wyrazów, jest **Term Frequences (TF)-Inverted Document Frequencies (IDF)**. Metoda ta dostarcza statystyki opisujące ważność słowa w dokumencie w odniesieniu do zbioru dokumentów i określana jest na podstawie współczynnika unikatowości danego wyrażenia. Obliczone w ten sposób wagi mają wartości rosnące proporcjonalnie do liczby wystąpień poszczególnych słów w dokumencie i malejące proporcjonalnie do jego występowania w całym zbiorze dokumentów. W ten sposób dostarczana jest informacja o częstości występowania słowa w dokumencie oraz w całym korpusie językowym. Dzięki temu słowo występujące rzadko w danym dokumencie, pomimo iż może być popularne w wielu dokumentach, w kontekście danego dokumentu zostanie oznaczone jako ważne. Aby obliczyć wagi należy określić częstotliwość występowania danego słowa w konkretnym dokumencie (TF) oraz sprawdzić jak często występuje ono we wszystkich dokumentach danego korpusu językowego (IDF). Oba współczynniki obliczamy za za pomocą poniższych wyrażeń:

$$tf_{ij} = \frac{n_{i,j}}{n_{k,j}} \tag{5.1}$$

$$idf_{ij} = log\frac{N}{m_i} \tag{5.2}$$

gdzie $n_{i,j}$ oznacza liczbę wystąpienia słowa w dokumencie *j*, $n_{k,j}$ liczbę wszystkich słów jakie ten dokument zawiera, *N* to liczba wszystkich dokumentów, natomiast m_i to liczba dokumentów zawierających co najmniej jedno wystąpienie słowa *i*. Mając wartości obu współczynników wagę słowa oblicza się za pomocą formuły:

$$tfidf_{i,j} = tf_{ij} * idf_{ij} \tag{5.3}$$

Popularne słowa mają niską wartość współczynnika *TF* oraz bliską zeru wartość współczynnika *IDF*, gdyż argument funkcji logarytmicznej jest większy lub równy 1, w związku z czym są odrzucane.

5.1.2.1. Model *N-gramów*

N-gram jest sekwencją *n* kolejnych wyrazów, bądź znaków, które można wyodrębnić ze zdania, stosowaną głównie do prognozowania kolejnych słów. W tym celu wykorzystuje się informację, iż pewne słowa bądź znaki występują często obok siebie. Przewidując *n*-te słowo, modele oparte na bazie *n*-gramów definicją warunkowe prawdopodobieństwo jego wystąpienia na podstawie *n-1* poprzednich słów. Prawdopodobieństwo to wyraża się następującą formułą:

$$P(x_1 \dots x_n) = P(X_1)P(X_2|X_1)P(X_3|X_1^2)\dots P(X_3|X_1^{n-1}) = \prod_{i=1}^{n} P(X_i|X_1^{i-1}) \tag{5.4}$$

W przypadku małych wartości *n* modele mają swoje konkretne nazwy i tak dla *n=1* **unigram**, dla *n=2* **bigram**, a dla *n=3* jest to **tigram**. Modele te należy stosować ostrożnie, gdyż dla dużego *n* staje się on kosztowny obliczeniowo.

5.1.3. Wektoryzacja tekstu za pomocą *Word embedding*

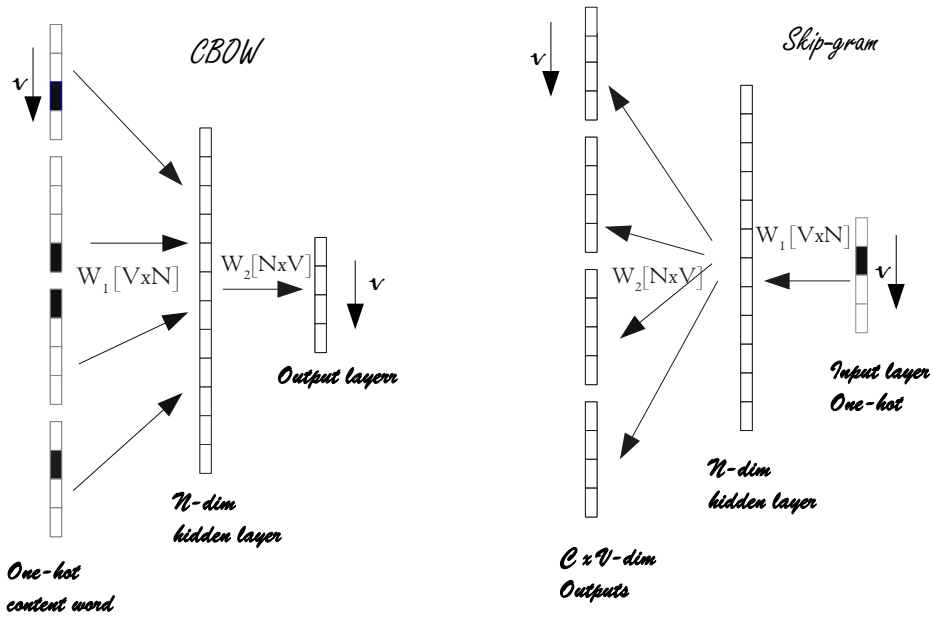
Word embedding jest numeryczną reprezentacją słów w postaci wektorów gęstych z niską liczbą wymiarów, w wysoce wymiarowej przestrzeni wektorowej. Słowa o podobnym znaczeniem semantycznym mają zbliżone wartości numeryczne - np. słowa *autor* i *publikacja* są w owej reprezentacji matematycznie bliżej niż *autor* i *kwadrat*. Ogólnie rzecz biorąc, geometryczna odległość między słowami odzwierciedla różnicę semantyczną między nimi. Podczas trenowania modelu pozyskiwana jest relacja pomiędzy słowem *w* i jego kontekstem *c*. Reprezentacje tego typu posiadają zdolność przeprowadzenia operacji arytmetycznych na słowach np. wynikiem operacji **vector('Paris') - vector('France') + vector('Italy')** będzie wektor bliski wektorowi **vector('Rome')**. Najpopularniejszymi modelami z *Word embedding* są Word2Vec [70] oraz GloVe [78].

5.1.3.1. Algorytm *Word2Vec*

Algorytm *Wor2Vec* jest dwu-warstwową siecią neuronową, przyjmującą na wejściu korpus tekstowy, zapisany w postaci wektora *one-hot encoding*. Kodowanie to polega na przypisaniu do każdego słowa unikalnego indeksu, pod którym w wektorze binarnym zostaje umieszczona wartość jeden, a pod wszystkimi innymi wartość zero. *Word2Vec* można zaliczyć do algorytmów typu *self-supervied*, które są specyficzną instancją algorytmów *supervised*, w których etykiety są generowane z danych wejściowych. Algorytm posiada dwie postacie: *Continous Bag of words (CBOW)* oraz *Skip-Gram*. Pierwsza z nich przewiduje słowo w danym kontekście, który może być pojedynczym słowem lub grupą słów, zwanym inaczej **oknem kontekstu**. Inaczej mówiąc dla danego kontekstu: w_{i-2}, w_{i-1}, w_{i+1}, w_{i+2}, model **CBOW** przewidzi słowo w_i. Natomiast model **skip-gram** dla danego słowa w_i przewidzi jego kontekst w_{i-2}, w_{i-1}, w_{i+1}, w_{i+2}. Można powiedzieć, że model **skip-gram** jest odwróceniem modelu **CBOW**, co w dobry sposób obrazuje rysunek (5.1).

W obu przypadkach każde słowo jest zakodowane w postaci *one-hot*, więc jeśli słownik ma rozmiar *V* na wejściu dostaniemy *V-wymiarowe* wektory (bądź wektor w zależności od modelu) z jedną wartością nie zerową. Jak pokazuje rysunek 5.1 między warstwą wejściową, a ukrytą znajduję się macierz wag W_1 o rozmiarach *VxN*, a między warstwą ukrytą a wyjściową umieszczona jest druga macierz wag W_2 o rozmiarach *NxV*, dzięki której możemy obliczyć wartość funkcji oceny dla każdego słowa w_j. Ponieważ wektor wejściowy jest w postaci *one-hot*, więc mnożenie wektora wejściowego przez macierz *W1* sprowadza się do wyboru odpowiedniego rzędu z macierzy W_1 co obrazuje rysunek 5.2. Wartość *N* oznacza liczbę neuronów warstwy ukrytej, która zarazem definiuje rozmiar wektora bądź wektorów wyjściowych.

W przypadku *CBOW* podczas treningu maksymalizowane jest prawdopodobieństwo warunkowe wystąpienia rzeczywistego słowa, zaobserwowanego dla danego kontekstu. Funkcją aktywacji *h* warstwy ukrytej są sumy wag tak zwanych *hot rows* dla danego słowa z macierzy*W1*, podzielone przez liczbę



Rysunek 5.1. Architektury word2vec

wejściowych wektorów.

$$h = \frac{1}{C} W(x_1 + x_2 + \cdots + x_c) = \frac{1}{C}(v_{w1} + v_{w2} + \cdots + v_{wc}) \tag{5.5}$$

Wartości te są przemnażane przez macierz *W2*, w celu wyliczenia funkcji oceny u_j dla każdego słwoa w_j:

$$u_j = {v'_{wj}}^T h \tag{5.6}$$

Aby wyznaczyć prawdopodobieństwo wystąpienia słowa w_j, pod warunkami wystąpienia danego kontekstu na wyjściu, użyta jest funkcja **Softmax**:

$$p(w_0|w_I) = \frac{exp(u_0)}{\sum_{i=1}^{V} exp(u_i)} \tag{5.7}$$

Wartość powyższego prawdopodobieństwa jest porównywana z rzeczywistym słowem, które winno się pojawić w danym kontekście, po czym następuję propagacja wsteczna, podczas której maksymalizowana jest funkcja straty w postaci:

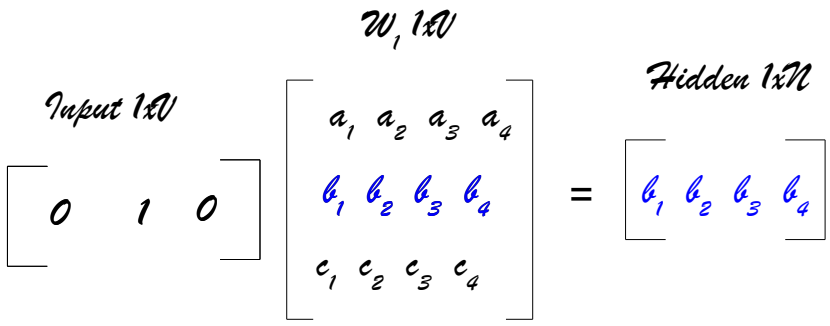
$$E = -log(p(w_o|w_i)) \tag{5.8}$$

Dla modelu **skip-gram** na wyjściu liczba prawdopodobieństw jest równa rozmiarowi kontekstu dla słowa w_t. W tym przypadku dla ciągu słów $w_1, w_2 \ldots w_T$ i kontekstu *C* maksymalizowana jest funkcja:

$$\frac{1}{T} \sum_{t=1}^{T} \sum_{j=-c}^{c} log(p(w_{t+j}|w_t)), j \neq 0 \tag{5.9}$$

Natomiast prawdopodobieństwa p(w_{t+j} | w_t) są zdefiniowane w następujący sposób:

$$p(w_0|w_I) = \frac{exp({v'_{w0}}^T v_{wI})}{\sum_{w=1}^{W} exp({v'_w}^T v_{wI})} \tag{5.10}$$



Rysunek 5.2. Mnożenie macierzy przez *one-hot* wektor

gdzie v_w oraz v'_w oznaczają wejściowe i wyjściowe reprezentacje wektora słowa *w*, a *W* jest liczbą słów w słowniku. W obu przypadkach aktualizowanie wektora dla każdego słowa wyjściowego jest operacją bardzo kosztowną. W związku z tym na wyjściu używa się **Hierachical Softmax** [72], który do reprezentacji każdego słowa występującego w słowniku używa drzew binarnych. W podejściu tym każde słowo reprezentuje liść, do którego z korzenia prowadzi unikalna ścieżka, która jest użyta do oszacowania prawdopodobieństwa dla danego słowa.

Oba modele mają swoje wady i zalety. Model *skip-gram* dobrze się sprawdza w pracy z mniejszymi zbiorami oraz rzadko z występującymi słowami. Z kolei CBOW jest szybszy i posiada lepszą reprezentację dla często występujących słów.

5.1.3.2. Model GloVe

Model *GloVe (Global Vector for word represenation)* w przeciwieństwie do przestawionego wyżej modelu *word2vec* nie ignoruje informacji o częstszym występowaniu pewnych kontekstów słów od innych. W tym celu z treningowego korpusu budowana jest macierz współwystępowania *X*, gdzie wartość X_{ij} oznacza liczbę wystąpień słowa *j* w kontekście słowa *i*, $X_i = \sum_{k=1}^{V} X_{ik}$ jest liczbą wszystkich słów jakie pojawiły się w kontekście słowa *i* (*V* - wielkość słownika), natomiast $P_{ij} = \frac{X_{ij}}{X_i}$ jest prawdopodobieństwem pojawienia się słowa *j* w kontekście *i*. W celu rozróżnienia bardziej znaczących słów od mniej ważnych, autorzy opisywanego modelu [78] proponują zamiast sprawdzania czystego prawdopodobieństwa możliwości współwystępowania słów, obliczyć stosunek prawdopodobieństw współwystąpień słów, posługując się poniższą formułą:

$$F(w_i, w_j, \widetilde{w_k}) = \frac{P_{ik}}{P_{jk}} \tag{5.11}$$

gdzie *w* są wektorami słów, natomiast $\widetilde{w}$ są oddzielnymi wektorami kontekstu słowa. Wartość powyższego stosunku dostarcza informację o wzajemnej relacji między próbnym słowem *k* a słowami *i* oraz *j* - gdy jego wartość jest wysoka wówczas słowo *k* jest mocno związane ze słowem *j* oraz w mniejszym stopniu ze słowem *i*. W równaniu (5.11) po lewej stronie występuje nieznana funkcja *F*, która jako argumenty przyjmuje wektory. Możliwości wyboru funkcji *F* jest wiele. W celu zachowania linowych relacji pomiędzy wektorami w_i, w_j oraz w_k, a tagże by przestrzec się przed mieszaniem wymiarów (prawa strona

równania jest skalarem) z argumentów liczona jest różnica oraz skalar, więc równanie (5.11) przyjmuje postać:

$$F((w_i - w_j)^T \widetilde{w_k}) = \frac{P_{ik}}{P_{jk}} \tag{5.12}$$

Dodatkowo relacja między słowem a kontekstem winna być symetryczna tzn. że ostateczny model powinien nie ulec zmianie po dokonaniu zamiany *w* -> $\widetilde{w}$ oraz *X* -> X^T stąd równanie (5.12) zapisuje się jako:

$$F((w_i - w_j)^T \widetilde{w_k}) = \frac{F(w_i^T \widetilde{w_k})}{F(w_j^T \widetilde{w_k})} \tag{5.13}$$

gdzie:

$$F(w_i^T \widetilde{w_k}) = P_{ik} = \frac{X_{ik}}{X_i} \tag{5.14}$$

Używając jako funkcji *F* funkcji eksponencjalnej, oraz wciągając wyrazy niezależne od *t* do parametru b_i otrzymuje się główne równanie modelu *GloVe*:

$$w_i^T \widetilde{w_k} + b_i + \widetilde{b_k} = log(X_{ik}) \tag{5.15}$$

Powyższe równanie traktuje wszystkie współwystąpienia słów jednakowo, co nie jest dobrym podejściem gdyż te rzadziej występujące mają tendencję do tworzenia zbędnego szumu. Z tego względu pożądane jest, aby te występujące częściej miały bardziej znaczące wagi. W tym celu do obliczania funkcji kosztu zostaje wprowadzana dodatkowa funkcja *f*, której zadaniem jest odpowiednie dopasowanie wag, uwzględniające częstotliwość współwystępowania słów. Gdy liczba współwystąpień jest większa niż ustalony próg, wtedy waga będzie miała wartość 1, w przeciwnym wypadku będzie mniejsza, co pokazuje poniższe równanie:

$$f(x) = \begin{cases} (\frac{x}{x_{max}})^\alpha, x < x_{max} \\ 1, otherwise \end{cases} \tag{5.16}$$

Ostatecznie używając powyższej funkcji, funkcja kosztu jest w postaci:

$$J = \sum_{i,j} f(X_{ij})(w_i^T \widetilde{w_j} + b_i + \widetilde{b_j} - logX_{ij})^2 \tag{5.17}$$

5.1.3.3. FastText

Jedną z wad opisanych wyżej algorytmów jest nieobsługiwanie słów spoza słownika. W tym celu Bojarowski i inni [4] zaproponowali rozwiązanie bazujące na modelu *skip-gram* (rozdział 5.1.3.1) w połączeniu z *n-gramami (rozdział* 5.1.2.1). W podejściu tym każde słowo reprezentowane jest jako zbiór znaków - *n*-gramów, które budowane są na bazie pod-słów, co pozwala modelowi na obliczenie reprezentacji dla słów, które nie występują w zbiorze treningowym. Każde słowo *w* jest reprezentowane jako zbiór *n*-gramów otoczonych specjalnym symbolem <> na początku i końcu słowa, oraz samego słowa *w*. Przykładowo niech *w=where*, a *n=3* wówczas przy takich założeniach otrzymuje się następujące *n-gramy*: **<wh, whe, her, ere, re>** oraz słowo **<where>**. Warto zauważyć, że dzięki temu otrzymana została reprezentacja dla słowa her, które w języku angielskim występuję bardzo często, a dzięki omawianemu podejściu nie musiało być one częścią zbioru treningowego. Mając słownik zbudowany z *n*-gramów o rozmiarze *G* oraz oznaczając jako $G_w \subset \{1,\ldots,G\}$ zbiór *n*-gramów pojawiających się w słowie *w*,

natomiast niech z_g będzie reprezentacją wektorową dla każdego *n*-gramu g, wówczas dane słowo jest reprezentowane jako suma wektorów zbudowanych z jego *n*-gramów, a funkcja celu przyjmuje postać:

$$s(w,c) = \sum_{g}^{G_w} z_g^T v_c \tag{5.18}$$

5.1.4. Model językowy

Metody z rodziny *embedings* są nie zależne od kontekstu co znaczy, że nie biorą pod uwagę kolejności słów w zdaniu czego wynikiem jest jedna wektorowa reprezentacja dla słowa niezależnie od kontekstu jego użycia. W związku z tym słowa, które mogą mieć wiele znaczeń (homonimy) jak na przykład *zamek* niezależnie od znaczenia w jakim zostały użyte będą reprezentowane przez ten sam wektor. Z tego względu powstała ulepszona forma reprezentacji tekstu, będąca świadoma kontekstu słowa zwana *modelem językowym*. Dzięki zastosowaniu modelów językowych poprawiano efektywność wielu kluczowych zadań z rodziny NLP takich jak: modelowanie języka, klasteryzację dokumentów, problem pytanie i odpowiedź QA (question and answer), czy semantyczną analizę języka LSA (ang. *Latent Semantic Analysis*). Na przestrzeni ostatnich lat powstało wiele modeli językowych, z których ważniejsze takie jak ELMO [79], Tranformers [117] oraz Bert [14] zostaną bardziej szczegółowo przedstawione.

5.1.4.1. Model *ELMo*

ELMo (ang. Embeddinggs from Language Models) jest przykładem kontekstowej reprezentacji słowa, budowanej na podstawie jego złożonej charakterystyki tj. składni oraz znaczenia semantycznego, jak również użycia słowa w różnych kontekstach czyli wieloznaczności. Model ten składa się z dwóch ukrytych warstw, które są dwu-kierunkowymi sieciami **LSTM** (rozdział 2.2.2), dzięki czemu model zna sens zarówno słów poprzedzających jak i następujących, co można nazwać modelem dwukierunkowym dalej nazywanym **biLM**. W modelu takim dla sekwencji *N* tokenów *(t_1, $t_2 \dots t_N$)*, obliczane jest prawdopodobieństwo tokenu t_k na podstawie poprzedzających go tokenów (propagacja w przód):

$$p(t_1, t_2, \dots t_N) = \prod_{k=1}^{N} p(t_k | t_1, t_2, \dots t_{k-1}) \tag{5.19}$$

jak również biorąc pod uwagę tokeny po nim następujące (propagacja wsteczna):

$$p(t_1, t_2, \dots t_N) = \prod_{k=1}^{N} p(t_k | t_{k+1}, t_{K2}, \dots t_N) \tag{5.20}$$

ELMo maksymalizując finalne prawdopodobieństwo dla słowa używa warstwy softmax oraz jako dwukierunkowy model językowy uwzględnia prawdopodobieństwa dla przedniej i wstecznej propagacji, co można zapisać jako:

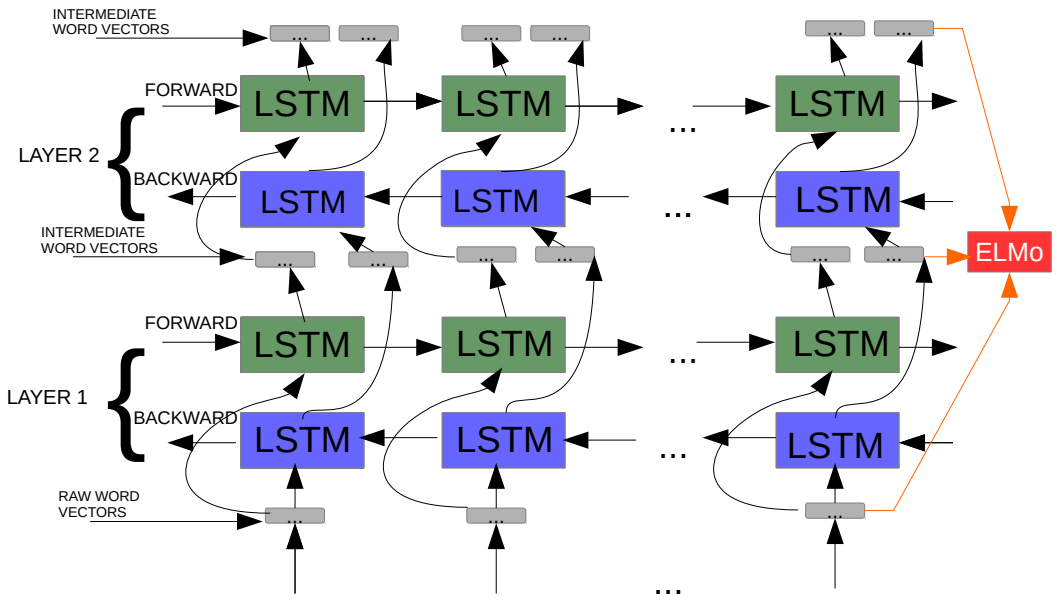
$$\sum_{k=1}^{N} (logp(t_k | t_1, \dots, t_{k-1}); \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s) + (logp(t_k | t_{k+1}, \dots, t_N); \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s)) \tag{5.21}$$

gdzie Θ_x jest reprezentacją tokenu, Θ_s jest rezultatem użycia warstwy Softmax, natomiast $\overrightarrow{\Theta}_{LSTM}$ oraz $\overleftarrow{\Theta}_{LSTM}$ są wyjściami warstw LSTM dla propagacji w przód i wstecz. Warto zauważyć, że zarówno

reprezentacje tokenów, jak i wartości warstw softmax, są dzielone pomiędzy oba modele, w przeciwieństwie do parametrów warstw LSTM (wewnętrzny stan, brama, pamięć), które są osobne. Autorzy rozwiązania zaproponowali, aby wejściowe tokeny w postaci x_k^{LM} były niezależne od kontekstu, czyli na przykład zbudowane za pomocą jednej z opisanych wcześniej metod *word-embedings*. Tokeny takie są przepuszczane poprzez *L* warstw *LSTM*, używając obu typów propagacji. Wówczas dla każdego tokenu t_k *L*-warstwowa sieć biLM oblicza $2L + 1$ reprezentacji R_k w postaci:

$$R_k = \{x^{LM_k}, \overrightarrow{h_{k,j}^{LM}}, \overleftarrow{h_{k,j}^{LM}} | j = 1, 2 \ldots, L\} \tag{5.22}$$

gdzie $\{\overrightarrow{\mathrm{h}_{k,j}^{LM}}, \overleftarrow{\mathrm{h}_{k,j}^{LM}}\}$ oznacza kontekstowo zależną reprezentację, będącą jednocześnie wejściem do kolejnych warstw biLM. Dodatkowo reprezentacje te tworzą tzw. *wektory pośrednie*, których kombinacja z wektorami wejściowymi stanowi wektorową reprezentację całego modelu ELMo, co obrazuje rysunek 5.3. Matematyczny opis ostatecznej reprezentacji przyjmuje postać $ELMo_k = E(R_k, \Theta_e)$. Mając wy-



Rysunek 5.3. Architektura ELMo

trenowany model językowy, w ostatnim kroku jest obliczenie finalnej reprezentacji słowa dla każdego zadania z osobna (klasteryzacja, analiza semantyczna, problemów pytanie-odpowiedź itp.), gdyż każde zadanie posiada specyficzne dla siebie parametry. W tym celu używa się formuły:

$$ELMo_k^{task} = E(R_k; \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} h_{kj}^{LM} \tag{5.23}$$

gdzie s_i^{task} reprezentuje znormalizowane wagi softmax, natomiast γ^{task} jest parametrem skalującym, specyficznym dla każdego zadania. Dzięki czemu wagi w warstwach ukrytych są zależne od zadania

do jakiego model będzie użyty, co podnosi skuteczność algorytmu. Krok ten jest nazywany *fine-tuning*, gdyż dopasowuje on maksymalnie parametry pod konkretne zadanie.

5.1.4.2. Model *Transformes*

Transformes jest modelem opartym na architekturze *encoder-decoder*, która przez lata była rozwijana przez wielu naukowców [66][126][12]. Modele tego typu przeznaczone są głównie do jednego z bardziej popularnych zadań z dziedziny przetwarzania języka naturalnego tj. problemu *sequence to sequence (seq2seq)*. Jak sama nazwa wskazuje, zadaniem *seq2seq* jest zamiana jednej sekwencji na inną np. jak ma to miejsce w tłumaczeniu z jednego języka na inny. Wykorzystywane do tego są zwykle dwie rekurencyjne sieci neuronowe, najczęściej *LSTM*, jedna będąca koderem oraz druga pełniąca rolę dekodera. W problemie zamiany jednej sekwencji na drugą często występuje problem iż sekwencja wejściowa i wyjściowa będą mieć różne długości (to samo zdanie w języku polskim zazwyczaj ma inną długość niż w angielskim), więc z punktu widzenia prawdopodobieństwa dla wejściowej sekwencji *(x_1, ... x_T)* model oblicza jej wyjściowy odpowiednik w postaci *(y_1, ... $y_{T'}$)* za pomocą funkcji : *p(x_1, ... x_T | y_1, ... $y_{T'}$)*, gdzie *T* i *T'* oznaczają kolejno długości obu tych sekwencji. Wejściem kodera są wektory o ustalonym rozmiarze, powstałe poprzez użycie jednej z wyżej opisanych metod *(word embedings)*. Wektory te czytane są przez sieć w sposób sekwencyjny, do momentu odczytania znaku kończącego oznaczonego jako *<EOS>*. Po każdym odczytaniu symbolu x_t, aktualizowany jest stan warstw ukrytych sieci, poprzez wyrażenie: $h_{<t>} = f(h_{<t-1>}, x_t)$, gdzie f oznacza funkcje aktywacji. Stan ostatniej warstwy ukrytej, reprezentuje całą wejściową sekwencję i jest zapisany w postaci wektora kontekstu **c**. Wektor kontekstu stanowi wejście do drugiej sieci rekurencyjnej (dekodera), który jest trenowany pod kątem generacji wyjściowej sekwencji, poprzez przewidywanie następnego tokenu y_t dla danego stanu warstwy ukrytej h_t. Obie te wartości są uwarunkowane wartością y_{t-1} oraz wektorem kontekstu c. Stąd wartość stanu warstwy ukrytej w czasie *t* oblicza się za pomocą :

$$h_t = f(h_{t-1}, y_{t-1}, c), \tag{5.24}$$

a rozkład prawdopodobieństwo dla następnego symbolu:

$$P(y_t|y_{t-1}, y_{t-2}, \dots y_1, c) = g(h_{<t>}, y_{t-1}, c) \tag{5.25}$$

gdzie f i g są funkcjami aktywacji (np. *Softmax*). Podczas treningu sieci rekurencyjnej typu *Koder-Dekoder* maksymalizowana jest następująca funkcja prawdopodobieństwa:

$$max \frac{1}{N} \sum_{n=1}^{N} log p_\theta(y_n|x_n) \tag{5.26}$$

gdzie θ jest zbiorem parametrów sieci, a para (x_n, y_n) jest kolejno wartością wejścia i wyjścia ze zbioru treningowego. Modele te sprawdzają się dobrze dla krótkich sentencji, jednak bardzo słabo radzą sobie z długimi, gdyż sieci RNN, nawet w przypadku LSTM, nie mogą pamiętać zbyt długich sentencji z powodu powszechnie znanego problemu zwanego *Vasing gradient problem* [77]. Dodatkowo informacja o wejściowej sekwencji jest skompresowana w postaci jednego wektora kontekstu o ustalonym rozmiarze, pochodzącego z ostatniej warstwy ukrytej, stąd trudności sieci z poradzeniem sobie z długimi

sentencjami szczególnie, gdy wyjściowa sekwencja powinna być dłuższa niż ta pochodząca z treningowego korpusu. Rozwiązaniem tego problemu jest mechanizm **Attention**, dzięki któremu prócz wektora kontekstu do dekodera przekazywane są wartości wszystkich stanów wewnętrznych kodera, połączone w taki sposób by utworzyły wagi. Dzięki użyciu owych wag możliwe jest wyznaczenie słów, na które powinna być zwrócona "większa uwaga"(attention) podczas wyznaczania kolejnego słowa w sekwencji wyjściowej. Wektor kontekstu c_i dla wyjścia y_i jest generowany poprzez wyrażenie:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{5.27}$$

gdzie $alpas$ są współczynnikami mówiącymi jak dużo uwagi powinno być zwrócone na poszczególne słowa podczas generacji słowa *i*, które są obliczane przy użyciu Softmax za pomocą wyrażenia:

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T_x} exp(e_{ik})} \tag{5.28}$$

natomiast e_{ij} oznacza "podobieństwo"pomiędzy poprzednim stanem dekodera s_{i-1} (czyli wartość dla poprzedniego słowa) oraz sumy wszystkich ukrytych stanów kodera h_j i przedstawia się jako:

$$e_{ij} = a(s_{i-1}, h_j) \tag{5.29}$$

Autorzy *Transformersa* [117] zaproponowali użycie sześciu koderów oraz takiej samej ilości dekoderów oraz, co ciekawe, zrezygnowaniu z sieci typu RNN. Zamiast tego zostały użyte warstwy: **self-attention** oraz oraz sieci typu **feedforward** (rozdział 2). *Self-attention* jest mechanizmem w którym podczas obliczenia reprezentacji dla danego słowa brane są pod uwagę otaczające go słowa. Inaczej mówiąc, podobnie jak w wyżej opisanym algorytmie ELMO, ważną rolę w reprezentacji słowa odgrywa jego pozycja w zdaniu, dzięki czemu możliwe jego lepsze rozpoznanie jego znaczenia semantycznego. W tym celu *self-attention* oblicza dla każdego słowa trzy wektory: *Query(Q), Key(K), Value(V)*, poprzez przemnożenie wektora wejściowego (embeddings) z trzema wektorami wag, które są inicjalizowane w sposób losowy oraz aktualizowane podczas propagacji wstecznej. Dzięki zastosowaniu trzech macierzy możliwe jest rozdzielenie wartości danego słowa od jego pozycji, w związku z czym możliwe staje się lepsze zrozumienie jego znaczenia semantycznego. Pierwszym krokiem w mechanizmie *self-attention* jest obliczenie tzw. *score*, który mówi ile uwagi należy poświęcić na resztę słów z wejściowej sekwencji, gdy kodujemy słowo na danej pozycji. Wartość ta obliczana jest poprzez obliczenie wartości skalarnej wektorów *q* oraz *k* danego słowa. W przypadku pierwszego słowa z sekwencji, pierwszy *score* będzie wartością skalara q_1 oraz k_1, drugi będzie skalarem q_1 oraz k_2, *n-ty* będzie skalarem q_1 oraz k_n. Czynność ta jest powtarzana dla każdego słowa czyli n^2 razy (*n*-dł. sekwencji). Mając obliczone wartości *score*, kolejnym krokiem jest podzielenie ich wartości przez stałą (w przypadku tranformers jest to wartość $\sqrt{d_k}$, czyli przez pierwiastek z wartości pierwszego wymiaru macierzy *k,q,v*), a następnie znormalizowanie ich wartości poprzez użycie funkcji Softmax. Otrzymane wartości są miarą ważności słowa na danej pozycji. Następnie wartości te są przemnażane przez wartość trzeciego z wektorów *value*, po czym z otrzymanego wyniku obliczana jest suma, której wynik jest wyjściem *self-attention*. Proces ten można zapisać za pomocą wyrażenia:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{5.30}$$

Bardzo ważnym usprawnieniem jakie zostało zastosowane w *Transformes* jest użycie tzw. *multihead attention*. Rozwiązanie to polega na powieleniu wyliczenia *self-attention n-razy* (w przypadku *Transformes* n=8), dzięki czemu model uczy się oddzielnie różnych rodzajów przydatnych informacji semantycznych, poprzez użycie różnych kanałów nazwanych przez autorów *heads*. Inaczej mówiąc model ma możliwość skupienia się na różnych pozycjach, ponieważ mamy zbiór wektorów *k, q, v*. Rozwiązanie to wymaga więcej wag oraz musi wykonać o wiele więcej operacji macierzowych. Z matematycznego punktu widzenia jest to wciąż jedna operacja macierzowa, ale z jednym dodatkowym wymiarem. W rezultacie otrzymujemy *n* macierzy, które zostają skonwertowane w jedną macierz poprzez, wymnożenie ich przez dodatkową macierz *W*, która również jest uczona podczas treningu. Zabieg ten jest konieczny, gdyż w *Transformes* po warstwie *self-attention*, zastosowana jest warstwa *feedforward*, która na wejściu oczekuje jednej macierzy. Ostatecznie wartość wyjściowa dla *Multihead attention* obliczana jest według wzoru:

$$MultiHead(Q, K, V) = Concat(head_1, \dots head_h)W^o \tag{5.31}$$

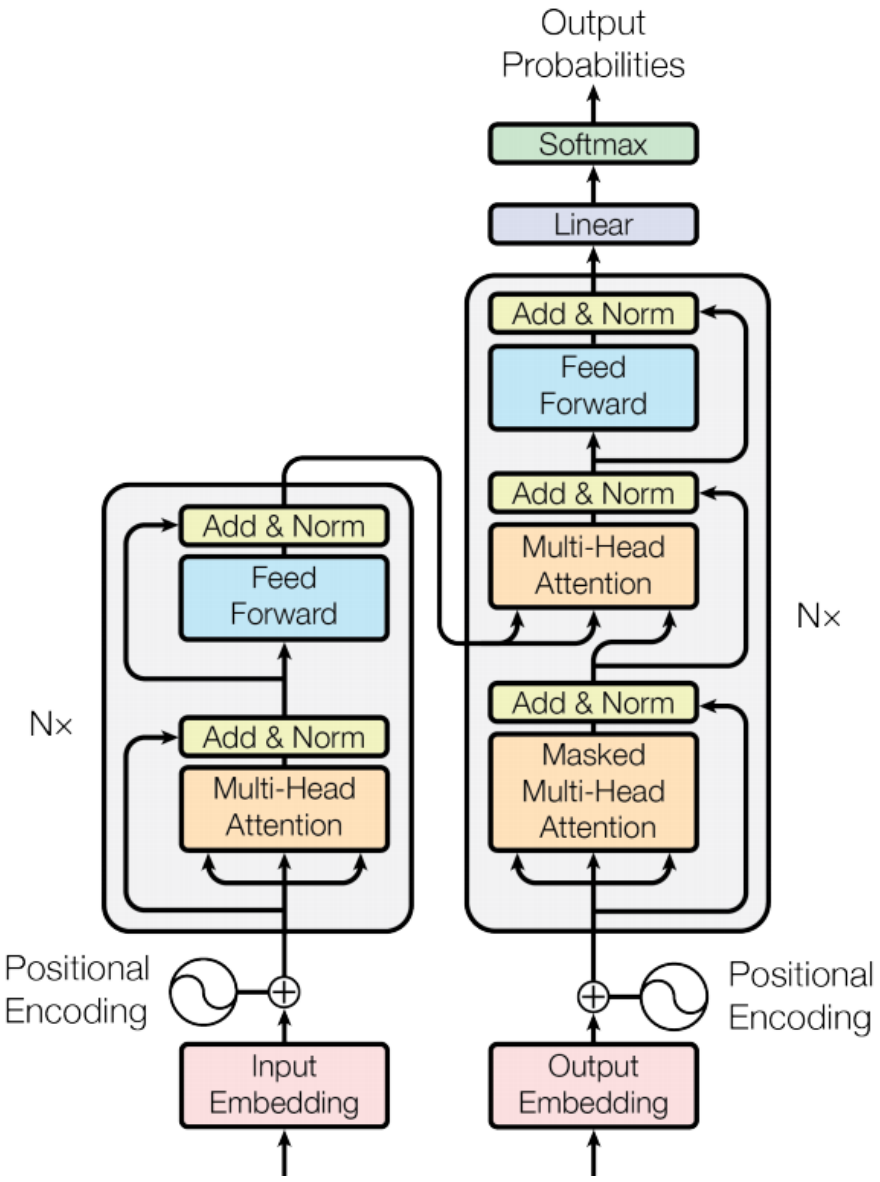
gdzie

$$head_i = Attention(QW_i^Q, KW_i^k, VW_I^V) \tag{5.32}$$

Strukturę, zarówno kodera jak i dekodera, modelu *Transformers* najlepiej obrazuje rysunek 5.4 pochodzący z oryginalnej publikacji na jego temat [117]. Ponieważ w modelu Transformers nie jest używana ani sieć rekurencyjna ani konwolucyjna, stąd brak jest informacji o pozycji danego tokenu w sekwencji. Jako rozwiązanie, autorzy zaproponowali użycie tzw. *positional encoding* widocznego na rysunku 5.4. Rozwiązanie to polega na wprowadzeniu dodatkowego wektora, o wymiarach takich samych jak wektory embedings, posiadającego tę informację. Wartości przechowywane przez *positional encoding* wektor są dodawane do wartości pochodzących z warstwy embdedings, a następnie są przesyłane do wejściowej warstwy *Transformers*. Istnieje wiele możliwości do zbudowania omawianego wektora. Autorzy modelu zaproponowali użycie funkcji trygonometrycznych, dzięki czemu możliwe jest odzwierciedlenie relatywnej pozycji tokenu w sekwencji [117].

5.1.4.3. Model *Bert*

Bert czyli *Bidirectional Encoder Representations from Transformers*, jak sama nazwa wskazuje, jest modelem językowym, który używa wcześniej opisanego modelu *Transofmers*, dzięki czemu model uczy się relacji miedzy słowami bądź pod-słowami w tekście. Głównym celem modelu *Bert* jest wygenerowanie modelu języka (podobnie jak miało to miejsce w przypadku ELMO (5.1.4.1)), stąd w modelu tym jedynie mechanizm kodera z architektury *Transformers*, jest wykorzystywany. Podczas treningu opisywany model nie czyta sekwencji od lewej do prawej, bądź od prawej do lewej, jak miało to miejsce w przypadku modelu *ELMO*. Zamiast tego *Bert* używa dwóch strategii z grupy metod bez nadzoru. Pierwsza z nich zwana **Masked LM (MLM)** polega na zastąpieniu pewnej liczby wejściowych, losowych tokenów (autorzy rozwiązania zaproponowali 15%), poprzez [MASK] token, a następnie próbie przewidzenia wartości oryginalnej, zamaskowanego tokenu, bazując na dostarczonym kontekście nie zamaskowanych tokenów. Dostarczony kontekst może dotyczyć zarówno tokenów poprzedzających jak i następujących po zamaskowanym, co pozwala na osiągniecie modelu dwu-kierunkowego. Z technicz-



Rysunek 5.4. Architektura Transofrmers [117]

nego punktu widzenia wymaga to dodania warstwy klasyfikatora na wyjściu kodera, która będzie obliczać prawdopodobieństwo dla każdego słowa ze słownika przy użyciu funkcji *Softmax*. Funkcja straty bierze tylko pod uwagę przewidywanie zamaskowanych słów. Druga ze wspominanych strategii nosi nazwę *Nex Sentence Prediction (NSP)*. Jej głównym zadaniem jest nauczenie modelu lepszego rozumienia zależności między dwoma zdaniami. Zdolność ta jest bardzo przydatna przy rozwiązywaniu takich zadań jak *Natural Language Interface (NLI)* oraz *Question Answering (QA)*. W tym celu wybierane są dwie losowe sentencje *A* oraz *B*, gdzie 50% sekwencji *B*, jest sekwencją następującą po *A* i jest oznaczana jako *IsNext*, a pozostała część sekwencji *B* jest wybierana losowo z korpusu i jest oznaczana jako *NoNext*. Aby było to możliwe, przed pierwszą sentencją wstawiany jest token [CLS], a na zakończeniu każdej z sekwencji wstawiany jest token [SEP]. Następnie wykonywany jest tzw. *sentence embedding*, który działa podobnie jak wcześniej opisany *word embedding*, z tym że w tym przypadku tokenami są

całe zdania. W kroku tym do każdego tokenu dodawana jest informacja, którego wcześniej wspomnianego zdania *A* czy *B* jest on częścią. Ostatnim krokiem jest uruchomienie *positional embeding*, który został opisany w rozdziale o *Transformers* (rozdział 5.1.4.2). Podczas *NSP* obliczane jest prawdopodobieństwo *IsNextSequence* przy pomocy funkcji *Softmax*. W czasie treningu minimalizowana jest funkcja straty jako kombinacja *LM* i *NSP*. Warto podkreślić, że dla modelu *Bert*, tokenizacja nie odbywa się na słowach ale na pod-słowach. Aby używać modelu *Bert*, podobnie jak miało to miejsce w *ELMO*, w celu osiągnięcia lepszych rezultatów, w wielu zadaniach z grupy przetwarzanie języka naturalnego, należy wykonać ostatnią fazę treningu czyli tzw. *Fine-tuning*. W zależności od zadania do jakiego będzie model przeznaczony, należy wykonać jeden z następujących kroków :

1. klasyfikacja dokumentów, realizowana jest podobnie jak klasyfikacja następnej sentencji (NSP), z tym, że na wyjściu dodawana jest warstwa klasyfikacji,

2. w przypadku zadania *QA*, model otrzymuje pytanie dotyczące tekstu oraz odpowiednią dla niego odpowiedź, co może być zrealizowane przy pomocy dodatkowych dwóch wektorów,

3. *NER* model otrzymuje sentencje, a na wyjściu zwraca różne typy jednostek,jakie w tekście się pojawiły, jak np. *osoba, firma, organizacja* itp. Aby było to możliwe, wykorzystuje się warstwę klasyfikacyjną, która uczona jest w ten sposób by mogła przewidzieć etykietę *NER*.

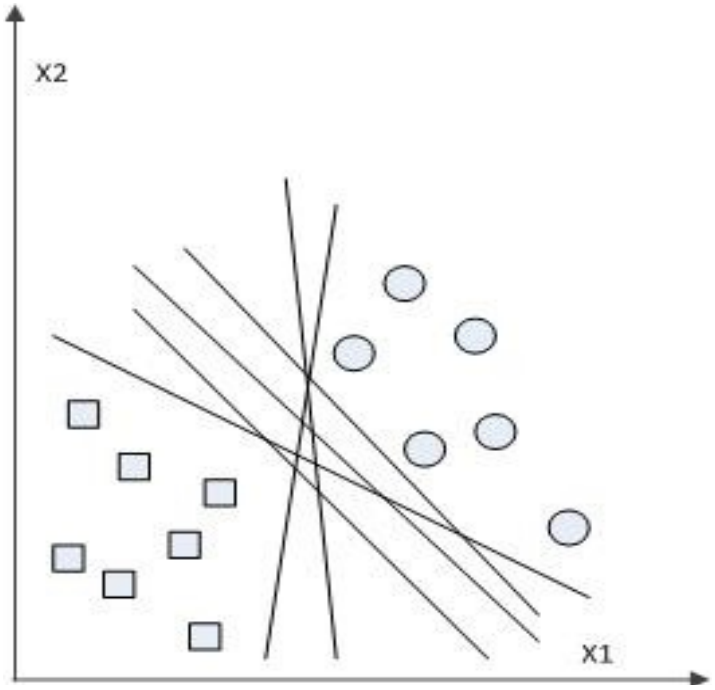
W fazie *fine-tuning*, większość parametrów, pozostaje taka sama jak w fazie treningu. Autorzy rozwiązania, dostarczają dokładnych wskazówek jak należy wykonać tę fazę, aby uzyskać jak najwyższą skuteczność algorytmu.

5.2. Algorytm wektorów nośnych - Support Vector Machines

5.2.1. Ogólny opis metody

Metoda wektorów nośnych SVM (ang. *Support Vector Machines*), została zaproponowana przez Vladimira Vapnika [13]. SVM jest klasyfikatorem binarnym, co oznacza że przy jego użyciu możliwe jest zaklasyfikowanie obiektu do jednej z dwóch klas. Podstawą metody jest odnalezienie maksymalnej płaszczyzny separującej obiekty, która pełni rolę kryterium decyzyjnego o przynależności danego obiektu do konkretnej klasy. Proces uczenia klasyfikatora, czyli inaczej mówiąc szukanie maksymalnej płaszczyzny separującej, jest przykładem uczenia z nadzorem (rozdział 2), czyli dla nowych danych wejściowych jest podpowiadana pożądana wartość wyjścia, więc należy mieć do dyspozycji dane oznaczone. W przypadku algorytmu wektorów nośnych dane wejściowe traktowane są jako wektory reprezentujące współrzędne punktów w przestrzeni N-wymiarowej, w której budowana jest hiperpłaszczyzna zdolna do segmentacji danych należących bądź nie należących do szukanego wzorca. W przypadku obrazów cechami wejściowych wektorów mogą być elementy pochodzące z wektora kolorów RGB, a jeśli chodzi o tekst są to w zależności od użytego algorytmu mapowania, liczbowe reprezentacje pojedynczych bądź grupy słów (rozdział 5.1.3). Jak można zauważyć na rysunku 5.5, możliwe jest poprowadzenie wielu płaszczyzn separujących dane. Nadrzędnym problemem jest wybór tej, która rozdziela dane przypisane do dwóch

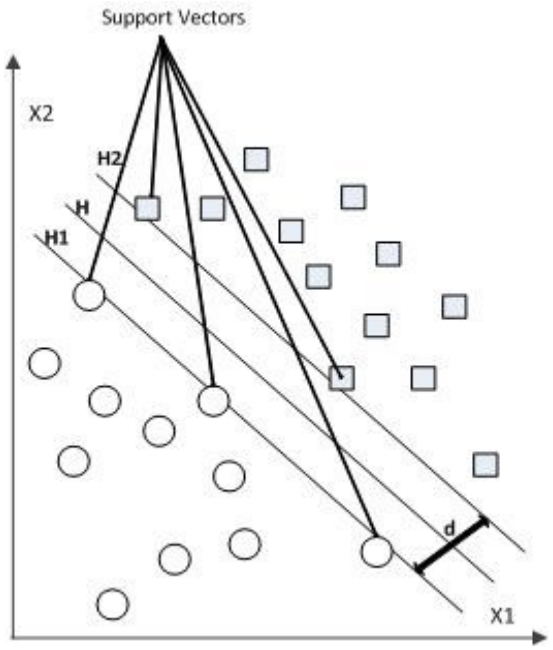
różnych klas z największym możliwym marginesem. Stąd też często opisywany klasyfikator jest nazywany *klasyfikatorem maksymalno-odległościowym*. Wspomniany margines jest definiowany jako maksymalna odległość między najbliższymi pozytywnymi i negatywnymi próbkami danych. Punkty położone najbliżej hiperpłaszczyzny separującej są nazwane **Wektorami Nośnymi** (ang. *Support Vectors - SVs*). Obliczanie wektorów nośnych może odbywać się dla danych separowalnych liniowo (rysunek 5.6), których przykładem są dane tekstowe, lub takich dla których rozdział za pomocą lini prostej jest niemożliwy - dane nie separowlane liniowo (rysunek 5.7), jak na przykład obrazy. Główną zaletą klasyfikatora SVM jest zdolność jego wytrenowania w oparciu o margines separacji między danymi, a nie o liczbę cech. W porównaniu z sieciami neuronowymi SVM potrzebuje mniej danych uczących do osiągnięcia wysokiej skuteczności klasyfikacji oraz jest bardziej odporny na przetrenowanie (ang. *overfitting*) [21]. Metodę wektorów wspierających stosuje się zarówno do klasyfikacji obrazów [48][59][82], kategoryzacji tekstów [47][114][124] jak również regresji [27][49][60].



Rysunek 5.5. Przykłady wielu możliwych płaszczyzn separujących dane

5.2.2. Wstęp do opisu matematycznego

W przypadku SVM dane uczące reprezentowane są w postaci par $(x_1, y_1),\ldots,(x_n, y_n)$, gdzie x_i jest k-wymiarowym wektorem (k-ilość cech), natomiast y_i przyjmuje wartość +/- 1 i jest informacją o przynależności (wartość +1) bądź jej braku (wartość -1) i-tego wektora do szukanego obiektu.



Rysunek 5.6. Przykład liniowej płaszczyzny separującej

5.2.3. Matematyczny opis liniowego klasyfikatora SVM

Liniowy klasyfikator SVM separuje dane w *n*-wymiarowej przestrzeni przy użyciu płaszczyzny decyzyjnej, definiowanej przy pomocy wyrażenia :

$$f(x) = w^T x + b \tag{5.33}$$

gdzie $w \in \Re^n$ jest *Wektorem normalnym płaszczyzny* (wektor prostopadły do płaszczyzny lub w przypadku innych powierzchni prostopadły do płaszczyzny stycznej do powierzchni w danym punkcie) [106], natomiast *x* jest wektorem wejściowym. Odległość między powierzchnią, a początkiem układu współrzędnych, definiuje zależność $b/\|w\|$, gdzie wyrażenie $\| \odot \|$ oznacza normę euklidesową wektora *w*, $b \in \Re$. Wektory *w* oraz współczynniki *b* winny tak definiować hiperpłaszczyznę, by odległość między najbliższymi punktami dwóch różnych klas była jak największa. Dla danych separowalnych liniowo musi być spełniony następujący warunek:

$$y_i(w^T x_i + b) - 1 \geq 0, y_i \in \{-1, 1\} \tag{5.34}$$

Równanie 5.34 opisuje dwie równoległe płaszczyzny, zawierające punkty z najmniejszym marginesem separacji dla optymalnej płaszczyzny separującej, czyli wektory nośne - SVs. Odległości między płaszczyznami H1, H2 (rys. 5.6), a początkiem układu współrzędnych wynosi odpowiednio : $|1-b|/\|w\|$ oraz $|-1-b|/\|w\|$. Bazując na tym obliczana jest odległość *d* pomiędzy dwoma równoległymi płaszczyznami :

$$d = 2/\|w\| \tag{5.35}$$

Głównym celem jest maksymalizacja marginesu separującego próby uczące, więc wartość $\|w\| = \sqrt{w^T w}$ powinna być minimalizowana, co jest równoznaczne z minimalizacją wyrażenia :

$$\min_{w,b} = \frac{\|w\|^2}{2} \tag{5.36}$$

Problem ten rozwiązuję się przy pomocy tzw. funkcji Langrange'a [113]. Oznaczając mnożniki Langrange'a jako α_i, opisywany problem optymalizacyjny można zapisać jako :

$$L(w, b, \alpha) = \frac{\|w\|^2}{2} - \sum_{i=1}^{n} \alpha_i y_i (w^T x_i + b) + \sum_{i=1}^{n} \alpha_i \tag{5.37}$$

gdzie α_i spełniają warunek :

$$\alpha_i \geq 0, \forall i \tag{5.38}$$

Rozwiązanie problemu (5.37) uzyskuje się poprzez minimalizację funkcji *L* względem *w* oraz *b* wraz z jej maksymalizacją względem wszystkich wartości α_i, co jest znane jako warunki Karush-Kuhn-Tucker (KKT) [3], które są opisane następująco :

$$\begin{cases} \frac{\partial}{\partial w} L(w, b, \alpha) = w - \sum_{i=1}^{n} \alpha_i y_i x_i = 0 \\ \frac{\partial}{\partial b} L(w, b, \alpha) = -\sum_{i=1}^{n} \alpha_i y_i = 0 \end{cases} \tag{5.39}$$

Z równań (5.37) oraz (5.39) otrzymuje się funkcje Langrange'a znaną jako *dualne zadanie problemu Lagrangian* :

$$L(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^T x_j \tag{5.40}$$

Powyższą funkcję należy zmaksymalizować względem mnożników Langrange'a przy ograniczeniu (5.38). Dodatkowo musi być spełniony warunek Karush-Kuhn-Tucker definiowany jako :

$$\alpha_i (y_i (w^T x_i + b) - 1) = 0, \forall i \tag{5.41}$$

Powyższy warunek jest spełniony tylko dla wektorów podtrzymujących, z czego wynika, że jedynie w tym przypadku wartości mnożników mogą być dowolne, dla pozostałych przypadków winny być równe zeru. Rozwiązaniem opisanego problemu jest wzór na optymalną hiperpłaszczyznę :

$$w = \sum_{i=1}^{N_s} \alpha_{si} y_{si} x_{si} \tag{5.42}$$

gdzie: s jest przynależnością do zbioru wektorów podtrzymujących, N_s jego liczebnością.

5.2.4. Matematyczny opis nieliniowego klasyfikatora SVM

W rzeczywistości rzadko występują dane, które można odseparować za pomocą prostych. W celu klasyfikacji wzorców tego typu stosuje się pewne poluzowania ograniczeń z równania (5.34), poprzez wprowadzenie dodatkowych parametrów ξ oraz C, które dopuszczają przekroczenie przez niektóre

punkty granicy separacji, czyli inaczej mówiąc dopuszczają pewien błąd klasyfikacji. Wprowadzając oba te parametry otrzymujemy równanie:

$$\min_{w,b,\xi} = \frac{\|w\|^2}{2} + C\sum_{i=1}^{n}\xi_i \tag{5.43}$$

z uwzględnieniem :

$$y_i(w^T x_i + b) \geq 1 - \xi_i \tag{5.44}$$

gdzie $\xi_i > 0$, są to tzw. zmienne dopełniające, definiujące dopuszczalny margines błędu, który jest regulowany przy użyciu parametru C. Mając zdefiniowane nowe zmienne równanie (5.38) przyjmuje postać:

$$0 \leq \alpha_i \leq C, \forall i \tag{5.45}$$

Ostatecznie optymalne wartości wektora *w* są otrzymywane za pomocą równania :

$$\sum_{i=1}^{Ns}\alpha_{si}y_{si}x_i \tag{5.46}$$

Z uwzględnieniem warunków KKT (5.39) :

$$\sum_{i=1}^{Ns}\alpha_{si}y_{si} = 0 \tag{5.47}$$

równanie obliczające mnożniki Langrange'a jest w postaci :

$$L(w,b,\alpha) = \frac{\|w\|^2}{2} + C\sum_{i=1}^{n}\xi_i - C\sum_{i=1}^{n}\alpha_i[y_i(w^T x_i + b) - 1 + \xi_i] - \sum_{i=1}^{n}\mu_i\xi_i \tag{5.48}$$

gdzie μ_i wymusza dodatnie wartości ξ_i. Ostatecznie funkcja decyzyjna wygląda następująco :

$$f(a) = sgn(\sum_{i=1}^{n}\alpha_i y_i x_i^T a + b) \tag{5.49}$$

SVMs może być użyty do klasyfikacji danych nieseparawolanych liniowo, przy użyciu tzw. *kernel trick*, który powoduje podniesienie wymiarowości, w wyniku czego uzyskuje się krzywoliniową granicę klasyfikacji. Mapowanie z dwu-wymiarowej do wielo-wymiarowej przestrzeni dokonuje się poprzez transformację ϕ, która dokonuje przekształcenia : $\overrightarrow{x_i}$ -> $\phi(\overrightarrow{x_i})$. Wówczas iloczyn skalarny wektorów przyjmuje postać:

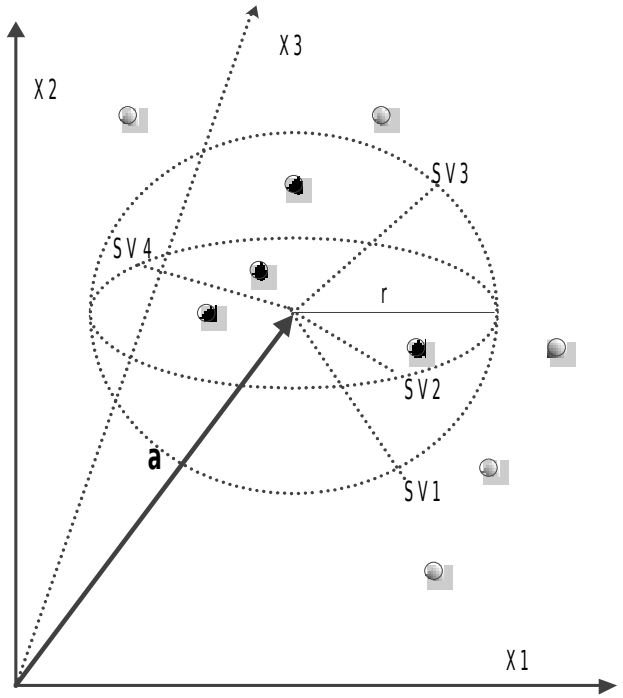
$$K(x_i, x_j) = \phi(\overrightarrow{x_i})^T\phi(\overrightarrow{x_j}) \tag{5.50}$$

gdzie *K* jest funkcją jądra $\Re^n$x$\Re^n$ -> $\Re$. Najczęściej używanymi funkcjami jądra są:

- liniowe : $K(x_i, x_j) = x^T x$
- wielomianowe rzędu : *d* $K(x_i, x_j) = (x_i^T x_j + 1)^d$
- tangens hiperboliczny : $K(x_i, x_j) = tanh(\gamma x^T x + C)$
- Gaussowskie : $K(x_i, x_j) = exp(-\gamma\|x_i - x_j\|^2)$

Po zastosowaniu jednej z funkcji jądra, funkcja decyzyjna przyjmuje postać:

$$f(a) = sgn(\sum_{i=1}^{n}\alpha_i y_i K(x_i^T a) + b) \tag{5.51}$$



Rysunek 5.7. Przykład nieliniowej płaszczyzny separującej [122]

5.2.5. Proces uczenia klasyfikatora binarnego

Głównym celem procesu treningu jest znalezienie takich par α_1, α_2, które spełniają kryteria opisane przez równania (5.45) i (5.47). Problem ten jest znany jako *kwadratowy problem optymalizacyjny* (ang. *quadratic programming optimization problem* - QP), który w przypadku SVM, rozwiązuję się poprzez algorytm zaproponowany przez John Platt [84] tzw. *Sequential Minimal Optimization (SMO)*. SMO jest iteracyjnym algorytmem, dekomponującym duży optymalizacyjny problem w wiele mniejszych pod-problemów, które następnie są rozwiązywane sekwencyjnie. W przypadku SVM algorytm działa na zbiorze **alfa**, ponieważ na podstawie tych wartości wyliczane są wartości wag **w**, a następnie uwzględniając wartość współczynnika **b**, obliczane są płaszczyzny separujące. Biorąc pod uwagę ograniczenie (5.47), mniejsze sub-problemy problemu QP angażują dwie alfy. W każdym kroku wybierane są α_1 oraz α_2, a następnie algorytm próbuje zoptymalizować ich wartości, po czym aktualizuje nimi wartości SVMs. Optymalizacja dwóch wybranych mnożników α_1, α_2 rozpoczyna się od sprawdzenia ograniczenia (5.47), które może być zapisane jako:

$$\alpha_1^{old} y_1 + \alpha_2^{old} y_2 + \sum_{i=3}^{Ns} \alpha_i^{old} y_i = 0 \tag{5.52}$$

Ponieważ optymalizujemy tylko α_1^{old}, α_2^{old}, więc ich nowe wartości α_1, α_2 muszą spełniać równanie :

$$\alpha_1 y_1 + \alpha_2 y_2 = \alpha_1^{old} y_1 + \alpha_2^{old} y_2 \tag{5.53}$$

oraz ograniczenie:

$$0 \leq \alpha_1, \alpha_2 \leq C \tag{5.54}$$

Wyznaczając α_1 z równania (5.53) izachowując ograniczenie (5.54) otrzymujemy:

$$0 \leq \alpha_1^{old} + y_1 y_2 \alpha_2^{old} - y_1 y_2 \alpha_2 \leq C \tag{5.55}$$

Na podstawie powyższego równania, dla α_2, wyznaczane są jego wartości graniczne L oraz H, takie że:

$$L \leq \alpha_2 \leq H \tag{5.56}$$

Uwzględniając powyższe nierówności oraz wiedząc, że wartości $y_1 y_2 \in \{-1, 1\}$, równania płaszczyzn L oraz H można zapisać jako:

– gdy $y_1 = y_2 => y_1 y_2 = 1$

$$L = max(0, \alpha_2^{old} - \alpha_1^{old} - C), H = min(C, \alpha_2^{old} + \alpha_1^{old}) \tag{5.57}$$

– gdy $y_1 \neq y_2 => y_1 y_2 = -1$

$$L = max(0, \alpha_2^{old} - \alpha_1^{old}), H = min(C, C + \alpha_2^{old} - \alpha_1^{old}) \tag{5.58}$$

Następnie uwzględniając powyższe ograniczenia, należy znaleźć optymalną wartość α_2, posługując się wyrażeniem:

$$\alpha_2 = \alpha_2^{old} + \frac{y_2(E_1 - E_2))}{\eta} \tag{5.59}$$

gdzie:

$$E_k = \sum_{j=1}^{m} \alpha_j y_j \langle x_j, x_k \rangle - y_k \tag{5.60}$$

$$\eta = 2\langle x_1, x_2 \rangle - \langle x_1, x_1 \rangle - \langle x_2, x_2 \rangle \tag{5.61}$$

Dokładne wyprowadzenie wzorów (5.59), (5.60), (5.61), dzięki którym możliwe jest określenie optymalnej wartości α_2 znajduje się [23]. Wartość E_k może być traktowana jako błąd wartości wyjściowej SVM, a *k*-tą wartością etykiety y_k. Podczas obliczenia wartości η w przypadku nieliniowym powinna być zastosowana funkcja jądra. Zamiana wartości α_2 odbywa się w następujący sposób:

$$\alpha_2^{new} = \begin{cases} H, \alpha_2 > H \\ \alpha_j, L \leq \alpha_2 \leq H \\ L, \alpha_2 < L \end{cases} \tag{5.62}$$

Ostatecznie, mając optymalną wartość α_2 obliczana jest optymalna wartość α_1:

$$\alpha_1^{new} = \alpha_1^{old} + y_1 y_2 (\alpha_2^{old} - \alpha_2^{new}) \tag{5.63}$$

Jako optymalizację procesu szukania par α_1 and α_2 Platt [84] zaproponował dla każdej z nich użycie dwóch różnych metod heurystycznych. W rezultacie czego, w opisanej optymalizacji, działają dwie pętle:

– zewnętrzna - przeznaczona do szukania α_2,

– wewnętrzna - dla wybranego α_1, poszukuje α_1

Na początku zewnętrzna pętla, iteruje po całym zbiorze treningowym, wybierając próbki, które nie spełniają warunku KKT (5.39) [87], oznaczając je jako kandydatów do optymalizacji. Po jednej takiej iteracji pętla zewnętrzna iteruje po próbkach, dla których mnożniki Langrange'a mają wartości różne od wartości granicznych: 0, C (są to tzw. *non-bound examples*). Podobnie jak uprzednio, podczas tej iteracji, sprawdzany jest warunek KKT, a próbki które go nie spełniają oznaczane są jako odpowiednie do optymalizacji. Warunkiem stopu iteracji po *non-bound examples* jest spełnienie przez wszystkie wartości tego zbioru warunku KKT, z uwzględnieniem błędu ξ (równanie (5.48)). Gdy to nastąpi, pętla zewnętrzna rozpoczyna ponowną iterację po całym zbiorze treningowym. W zewnętrznej pętli odbywają się na przemiennie dwie iteracje :

– pojedyncza - po całym zbiorze treningowym,

– wielokrotna - po zbiorze *non-bound examples*

Proces jest powtarzany tak długo, aż wszystkie wartości próbki zbioru uczącego będą spełniać warunek KKT z błędem ξ. W celach polepszenia działania w procesie uczenia zastosowano cache, którego zadaniem jest przechowywanie wartości błędu (E z równania (5.60)), dla każdej próbki ze zbioru *non-bound*. W momencie, gdy α_j jest wybrana, następuje wybór α_1, który dokonuje się poprzez maksymalizację wartości $|E_1 - E_2|$. W zależności od znaku E_1, w celu optymalizacji, algorytm wybiera próbkę z maksymalną (E_1 jest dodatnie) bądź minimalną (E_1 jest ujemne) wartością błędu E_2. W przypadku, gdy powyższe metody heurystyczne nie przynoszą znaczącego postępu, iteracja po zbiorze *non-bound examples* rozpoczyna się od nowa, w celu poszukiwania lepszych próbek. Gdy ten krok nie przynosi dalej zadowalającego rezultatu, rozpoczynają się na nowo poszukiwania próbek nie spełniających KKT w całym zbiorze treningowym, a następnie optymalizacja *non-bound examples*. Proces ten trwa tak długo, aż wszystkie próbki spełniają warunek KKT z pewną ustaloną dokładnością. Testy pokazały, że wystarczającą wartością dopuszczalnego błędu ξ jest 0.001. Oprócz mnożników α, w procesie treningu ustalana jest wartość parametru *b*. W algorytmie SMO wartość ta jest aktualizowana po każdym kroku, czyli gdy warunek KKT jest spełniony dla obu zoptymalizowanych próbek uczących α_1^{new}, α_2^{new} . Dla każdej z nowych wartości α^{new} obliczane są wartości b_1, b_2 za pomocą :

$$b_1 = E_1 + y_1(\alpha_1^{new} - \alpha_1)\langle x_1, x_1\rangle + y_2(\alpha_2^{new} - \alpha_2)\langle x_1, x_2\rangle + b \tag{5.64}$$

$$b_2 = E_2 + y_1(\alpha_1^{new} - \alpha_1)\langle x_1, x_2\rangle + y_2(\alpha_2^{new} - \alpha_2)\langle x_2, x_2\rangle + b \tag{5.65}$$

Nowa wartość b stanowi średnią arytmetyczną wartości b_1 oraz b_2. Podejście to zostało zaproponowane przez Vapnika [13].

$$b = \frac{b_1 + b_2}{2} \tag{5.66}$$

5.2.6. Metoda wektorów nośnych jako klasyfikator wielo-klasowy

Jak zostało wcześniej powiedziane, SVM jest klasyfikatorem binarnym (rozdział 5.2.1), co oznacza że problem klasyfikacji sprowadza się do podjęcia decyzji o przynależności obiektu do jednej z dwóch klas, oznaczonych zwykle jako +/-1. W celu użycia klasyfikatora binarnego do klasyfikacji *k* klas należy zastosować jedno z dwóch podejść:

I. **Jedna klasa przeciwko wszystkim innym** (ang. *One versus All - OVA*) - w tym podejściu problem dla *k* klas zastępowany jest *k* problemami dwuklasowymi. Każdy pojedynczy klasyfikator decyduje o przynależności do konkretnej klasy, oznaczanej jako +1, lub do (k-1) pozostałych klas, oznaczonych jako -1. Klasyfikator k_i odróżnia klasę Ω_i od zbioru klas $\{\Omega_0, \Omega_1 \ldots \Omega_{i-1}, \Omega_{i+1} \ldots \Omega_k\}$. W czasie klasyfikacji decyzja podejmowana jest przy użyciu formuły:

$$f_{ova}(x) = argmax_{i=1\ldots k} f_i(x) \tag{5.67}$$

II. **Jedna przeciw jednej** (ang. *One vers One - OVO*) - w tym przypadku budowanych jest *k(k-1)/2* klasyfikatorów binarnych - po jednym dla każdej pary klas (Ω_i, Ω_j), $i \neq j$. Każdy klasyfikator uczony jest na zbiorze próbek reprezentujących klasy *i* będące próbkami pozytywnymi, bądź *j* reprezentujące próbki negatywne. Następnie decyzja o przynależności do konkretnej klasy podejmowana jest na podstawie funkcji:

$$f_{ovo}(x) = argmax_{i=1\ldots k}(\sum_j f_{ij}(x)) \tag{5.68}$$

Klasyfikator ten często jest nazywany *wszyscy przeciw wszystkim* (ang. *All versus All - AVA*).

W obu przypadkach funkcja f(x) jest definiowana w zależności od typu klasyfikatora przez wyrażenie (5.49 - liniowy) bądź (5.51 - nieliniowy). W przypadku klasyfikatora typu OVO, istnieje potrzeba wyprodukowania większej liczby klasyfikatorów - wymaga on $O(n^2)$ zamiast $O(n)$ klasyfikatorów, jednak każdy z nich jest stosunkowo mały (otrzymuje niedużą porcję próbek uczących), więc klasyfikacja jest stosunkowo szybka. W niniejszej pracy zostało zastosowane podejście OVA, ze względu na mniejszą złożoność obliczeniową tego podejścia przy zachowaniu wysokiej skuteczności.

5.3. Kwantyzacja

Najnowsze architektury systemów uczenia maszynowego, głębokiego (rozdział 2) składają się z bardzo wielu warstw. Niektóre z nich mogą zawierać miliony parametrów i wymagać nawet do bilionów operacji matematycznych. Wytrenowanie takiego systemu może zająć do kilku tygodni. Wspomniana ilość koniecznych parametrów stanowi problem, szczególnie w przypadku akceleratorów sprzętowych takich jak układy GPGPU (rozdział 3.2), których użycie z kolei jest niezbędne w celu przyśpieszenia procesu uczenia. Powszechnie używaną techniką do akceleracji obliczeń, jak również ograniczenia zużycia pamięci, jest użycie w obliczeniach zredukowanej precyzji danych, co oznacza skonwertowanie liczby z jej *N*-bitowej do *n*-bitej reprezentacji, gdzie $N > n$. Transformacja redukująca precyzję liczby jest znana jako *Kwantyzacja*. Używając kwantyzację należy przede wszystkim zadbać o utrzymanie skuteczności

algorytmu. Oczywistością jest, że nie można dopuścić do znaczącego spadku efektywności algorytmu kosztem szybkości jego trenowania, czy też oszczędności wymaganej pamięci. W niniejszej pracy przyjęto założenie, iż spadek ten winien być maksymalnie $\sim 1\%$. Determinuje to minimalną liczbę bitów, przy użyciu których możliwe jest przeprowadzenie obliczeń bez generacji większej straty skuteczności algorytmu. Kluczową rolę odgrywa tutaj dobór metody kwantyzującej, odpowiednio do rodzaju algorytmu. Generalnie można rozróżnić dwa typy kwantyzacji tj. równomierną i nierównomierną. Do pierwszej grupy możemy zaliczyć te, w których jako funkcja mapująca zostało użyte dowolne przekształcenie liniowe, natomiast do kwantyzacji nierównomiernej te w których przy mapowaniu brały udział funkcje nieliniowe takie jak np. funkcja logarytmiczna czy też tangens hiperboliczny. W niniejszej pracy do wytrenowania algorytmu wektorów nośnych (rozdział 5.2) ze zredukowaną precyzją danych w celu klasyfikacji tekstu, zostały zaimplementowane dwie metody kwantyzujące. Przy użyciu zaproponowanych metod zbiór danych w formacie *single-precisoin* jest mapowany do jego reprezentacji w zredukowanej precyzji przez dwie stałoprzecinkowe kwantyzacje, wykorzystujące liniowe funkcje mapujące - kwantyzacja liniowa, których opis znajduję się w dalszej części rozdziału. Przykład zastosowania kwantyzacji liniowej i nieliniowej, w innych niż omawiane w tym rozdziale algorytmach, w dziedzinie sztucznej, inteligencji można znaleźć w pozycjach [80][120][125] .

5.3.1. Zmiennoprzecinkowy zapis liczby

Nieodłączną częścią wszelkich obliczeń naukowych są liczby zmiennoprzecinkowe, których zapis specyfikuje standard IEEE-754. Standard ten posiada bit wskazujący czy liczba jest ujemna/dodatnia - **S**, mantysę czyli bity, na których jest zapisana część ułamkowa - **F** oraz cechę - **E**, kodującą wykładnik potęgi liczby 2. Zgodnie ze standardem IEEE-754 liczba zmiennoprzecinkowa jest reprezentowana za pomocą wyrażenia:

$$(-1)^S \times F \times 2^E \tag{5.69}$$

Opisany standard jest nazwany zmiennoprzecinkowym, ponieważ liczba bitów przeznaczonych na mantysę może się zmieniać, stąd następuje przesunięcie przecinka, w przeciwieństwie do stałoprzecinkowego systemu reprezentacji, gdzie przecinek jest zawsze umieszczony w tym samym miejscu, co ogranicza zakres liczb jakie mogą być reprezentowane w tym zapisie. Standard IEEE-754 definiuje następujące formaty reprezentacji liczb:

- half-precision - E=5bits, F=10bits,
- single-precision - E=8bits, F=23bits,
- double-precision - E= 11bits, F=52bits,
- quaduple-precision - E=15bits, F=112bits.

Dodatkowo w standardzie tym zostały określone wartości specjalne, które mogą pojawić się jako rezultat niektórych niedozwolonych operacji zmiennoprzecinkowych takich jak np. dzielenie przez zero, czego wynikiem będzie nieskończoność. Dodatkowymi specjalnymi wartościami są m.in. NaN (Not a

Number), co oznacza niemożliwość określenia wartości wyniku, natomiast w zależności od znaku przekroczenie dopuszczalnego zakresu liczby (ang. Overflow) doprowadzi do wygenerowania plus/minus nieskończoności.

5.3.2. Metoda kwantyzująca typu *Max magnitude dynamic fixed-point*

Pierwsza zaproponowana metoda w celu reprezentacji liczby zapisanej formacie pojedynczej precyzji (float-point), w jej stałoprzecinkowym formacie : n_{total_bits}-*width*, musi zdecydować ile bitów będzie użytych do reprezentacji części całkowitej : $n_{integer\_part}$, a ile do części ułamkowej $n_{fractional\_part}$. Determinacja ilości bitów wymaganych do zapisu części całkowitej konkretnej liczby, odbywa się na podstawie jej maksymalnej bezwzględnej wartości, jaka może pojawić się w trakcie obliczeń. Na przykład, gdy maksymalną wartością dla liczby będzie 3, wówczas muszą być użyte minimum 2 bity do reprezentacji części całkowitej. W przypadku użycia opisywanej metody kwantyzującej do procesu uczenia algorytmu wektorów nośnych, statystyki determinujące maksymalną bezwzględną wartość danych liczb są budowane w tak zwanym procesie *pre-processingu* przy użyciu środowiska PYTHON [104]. Proces ten oblicza wszystkie wartości jakie są wejściem oraz wyjściem wszystkich pod-operacji użytych w procesie treningu SVM, a następnie spośród nich wybiera się największą bezwzględną wartość. Oczywiście w zależności od użytego zbioru treningowego, amplitudy wartości poszczególnych pod-operacji są różne, dlatego też gromadzenie statystyk zostało uruchomione dla każdego użytego zbioru z osobna. Posiadając maksymalne wartości poszczególnych liczb **X** można zastosować ogólny wzór na obliczanie minimalnej wartości $n_{integer\_part}$ w postaci:

$$n_{integer\_bits} = ceil(\log_2(max|X|) \tag{5.70}$$

na podstawie otrzymanej wartości, minimalna ilość bitów potrzebnych do zapisu części ułamkowej jest obliczana za pomocą wyrażenia :

$$n_{fractional\_bits} = n_{total\_bits} - n_{integer\_bits} - 1 \tag{5.71}$$

znając wartości: $n_{integer\_part}$ oraz $n_{fractional\_part}$ kwantyzacja liczby *X* jest przeprowadzona za pomocą wzoru:

$$q_{X_p} = 2^{-frac\_bits_p} \times round(2^{frac\_bits_p} \times X_p) \tag{5.72}$$

gdzie wyrażenie $2^{+/-(frac\_bits)}$ jest wartością przesunięcia bitowego wejścia X .

W celu przybliżenia działania metody warto rozważyć następujący przykład: chcemy zapisać liczbę *a* zapisaną na 32 bitach (typ *float*), używając tylko 8 bitów. Z zebranych statystyk wiadomo, że maksymalna bezwzględna jej wartość, jaka może pojawić się podczas procesu uczenia wynosi 3, z czego wynika, że liczba bitów przeznaczonych na zapis części całkowitej winna być $n_{integer\_part}$ = *2*. Pamiętając o bicie znaku, liczba bitów na której będzie zapisana części ułamkowej dla omawianego przykładu wynosi $n_{fractional\_part}$ = *5*. Niech w danym kroku poddana kwantyzacji będzie wartość : *a = 1.5738242232*. Wówczas wstawiając obliczone wartości do wzoru 5.72, na wyjściu otrzymany rezultat będzie wynosił *1.5625* i będzie on używany w kolejnych obliczeniach przetwarzanego algorytmu.

Wadą opisanego wyżej podejścia może być znaczna utrata informacji, w momencie dużego rozkładu danych. W badanym algorytmie przedstawione zjawisko może nastąpić, gdy np. wynikiem pod-operacji procesu uczenia SVM jest wartość, która może należeć do zbyt szerokiego przedziału.

5.3.3. Metoda kwantyzująca typu *Min-max dynamic fixed-point*

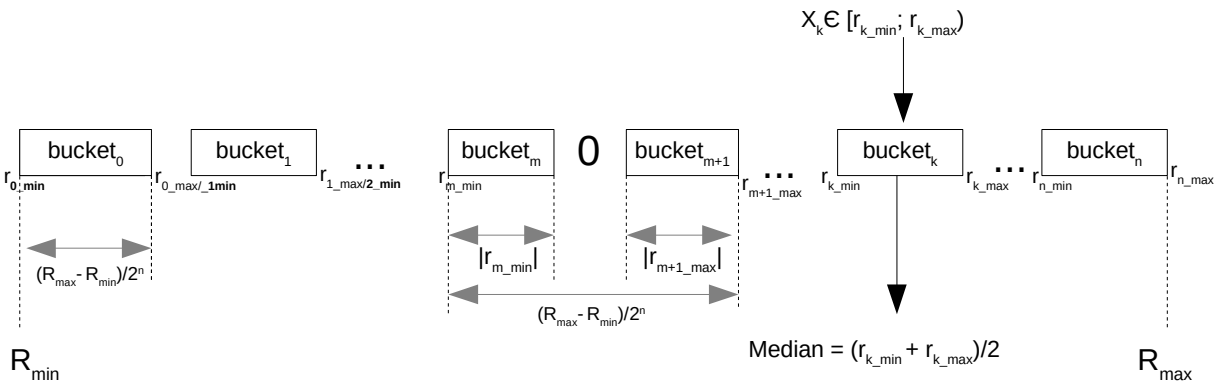
Druga zaproponowana metoda kwantyzacji, dla zbioru danych *S* typu pojedynczej/podwójnej precyzji (w niniejszej pracy do zapisu zbioru wejściowego użyto typu *float*), jako parametry wejściowe przyjmuje minimalną i maksymalną wartość R_{min} oraz R_{max} zbioru *S* oraz szerokość bitową $bits - width$, do jakiej ma zostać dokonane mapowanie. Posiadając parametry wejściowe, należy stworzyć *n* zbiorów o jednakowej szerokości, gdzie *n* jest wyjściową liczbą bitów funkcji mapującej, na jakiej będą reprezentowane liczby w procesie treningu SVM. Pojedynczy *i*-ty przedział budowany jest za pomocą formuły:

$$range_i = \left[r_{max_{i-1}}; r_{min_i} + \frac{R_{max} - R_{min}}{2^{Bitwidth}} \right) \tag{5.73}$$

gdzie $r_{min_i} = r_{max_{i-1}}$ oraz $r_{min_0} = R_{min}$.

Podobnie jak ma to miejsce we wcześniej opisanej metodzie (rozdział 5.3.2), na początku uruchamiany jest pre-processing, który gromadzi statystyki dla danego zbioru treningowego. W tym przypadku są to wartości R_{min}, R_{max}, jakie mogą wystąpić w każdej pod-operacji w czasie uczenia SVM. Każda pod-operacja *p* posiada swój przedział S_p, w jakim zawierają się jej dane wejściowe oraz wyjściowe. Kwantyzacja wartości X_p jest realizowana poprzez znalezienie zakresu $range_p$, ze zbioru zakresów S_p, a następne zastąpienie tej wartości przez **wartość mediany** przedziału $range_p$. W niektórych przypadkach omawiana kwantyzacja może zmienić znak kwantyzowanej liczby, co w efekcie może doprowadzić do znaczącego spadku efektywności algorytmu. W celu uniknięcia tego niepożądanego zjawiska należy zadbać o spełnienie warunku: $sign(r_{min_i}) = sign(r_{max_i})(r_{min_i}, r_{max_i}$ - składowe równianina 5.73). Warunek ten zapewnia zachowanie znaku w procesie kwantyzacji, czyli: $sign(r_{input\_quant_i}) = sign(r_{result\_quant_i})$. Założenie to, wymusza rozdzielenie zakresu zawierającego 0.f, czyli zakresu w postaci: $[r_{min_i} \ldots 0 \ldots r_{max_i})$, na dwa osobne zakresy w formie: $[r_{min_i} \ldots 0)$ i $(0 \ldots r_{max_i})$. Oba nowo utworzone w ten sposób zakresy nie zawierają wartości zero, ponieważ nie powinna ona być kwantyzowana, gdyż w następstwie mogłoby wywołać to niepożądany efekt.

W celu lepszego zilustrowania opisywanej kwantyzacji, warto dokonać analizy rysunku 5.8. Wspomniane wyżej przedziały S_p, dzięki którym odbywa się kwantyzacja, można przedstawić za pomocą kubełków (ang. *bucket*), do których w zależności od wartości, trafia poddawana kwantyzacji liczba. Na powyższym rysunku liczbą tą, jest X_k z przedziału $[r_{k\_min}; r_{k\_max})$, gdzie wartości graniczne przedziałów zostały obliczone przy użyciu formuły (5.73). Rezultatem kwantyzacji jest środkowa wartość użytego kubełka czyli średnia arytmetyczna jego wartości brzegowych. Kubełki r_m oraz r_{m+1} demonstrują sytuację, gdy w jednym z kubełków znalazła się wartość 0.f, a jak zostało to powiedziane, w takiej sytuacji należy rozdzielić taki kubełek na dwa odrębne zakresy. W rezultacie czego nowo powstałe kubełki, mają inne szerokości niż pozostałe, chyba że wartość 0.f w oryginalnym kubełku była wartością mediany.



Rysunek 5.8. Kwantyzacja metodą *Min-max dynamic fixed-point*

5.4. Implementacja

Proces implementacji zawiera trzy osobne moduły, z których dwa realizują tzw. *pre-processing* (rozdziały 5.3.2, 5.3.3), czyli przygotowanie wszystkich niezbędnych danych koniecznych do przeprowadzenia części głównej, realizującej proces uczenia klasyfikatora SVM, używając różnych szerokości danych. Jako zbiory treningowe zostały użyte: **The Reuters Dataset-r8** (Reuters articles with single label from R10 sub-collection of Reuters-21578) [108] oraz **WebKB** [109], które są zbiorami typu *multi-class* (jest wiele klas obiektów), jak również *multi-label* (każdy dokument może należeć jednocześnie do wielu klas). W zbiorach tych dokumenty są zgrupowane po klasach. Głównym celem treningu SVM jest obliczenie wektorów nośnych, które będą zdolne do klasyfikacji dokumentów. Jak zostało to nadmienione, SVM jest klasyfikatorem binarnym (rozdział 5.2.1), więc w celu realizacji klasyfikacji wieloklasowej został użyty algorytm **One-versus-All**, który został dokładnie opisany w rozdziale 5.2.6.

5.4.1. Kroki implementacji

W celu wytrenowania wektorów nośnych, dokonujących klasyfikacji dokumentów, zostały zaimplementowane następujące moduły:

I. **Zbieranie statystyk** - aplikacja ta została zaimplementowana w języku PYTHON i jest uruchomiana dla każdego zbioru treningowego z osobna, przed modułem realizującym uczenie. Jej zadaniem jest zgromadzenie informacji o największych i najmniejszych możliwych wartościach, jakie mogą się pojawić we wszystkich pod-operacjach w trakcie treningu SVM. Dla każdego z obliczeń pośrednich przechowywane są wartości skrajne jego wyniku, w postaci pary *Pairs<min, max>*. Po przeprowadzeniu danej operacji pośredniej, otrzymana wartość porównywana jest z jej dotychczasowymi rezultatami ekstremalnymi i w przypadku wystąpienia wartości większej/mniejszej od dotychczasowej maksymalnej/minimalnej, odpowiednie pole w parze *Pairs<min, max>* jest aktualizowane. Moduł ten został uruchomiony dla okrojonych zbiorów treningowych (r8, WebPk), gdyż

jest to wystarczające do uzyskania informacji koniecznych do zrealizowania kwantyzacji opisanych w rozdziałach 5.3.2,5.3.3.

II. **Obliczanie TF-IDF** - dane tekstowe zostają zapisane w formacie numerycznym za pomocą algorytmu TF-IDF (rozdział 5.1.2). Dodatkowo w tym kroku, dane wejściowe są oczyszczane poprzez:

   – zapis wszystkich wyrazów za pomocą małych liter,

   – usunięciu wszystkich tzw. *stopwords* czyli często używanych słów dla danego języka o małym znaczeniu jak np. spójniki, które nie wpływają na efektywność rozwiązywania problemów z zakresy przetwarzania języka naturalnego, takich jak np. klasyfikacja, analiza semantyczna itp.,

   – filtracja wyrazów składających się z mniej niż *n*-znaków (w niniejszej pracy *n* wynosi 4),

   – *steaming words* czyli obcięcie końcówki lub początku wyrazu z powszechnie używanych prefiksów oraz sufiksów, które można znaleźć w zmienionej formie słowa. Dzięki temu zabiegowi, słowa reprezentowane są poprzez ich rdzeń, np. słówko *asked*, zostanie zamienione na *ask*, a *studies* na *study*.

III. **Obliczanie parametrów SVM** - Obliczanie parametrów α oraz b (rozdział 5.2.1). Jest to główna część całego procesu trenowania wektorów wspierających. Jako wejście moduł przyjmuje dane zapisane w postaci macierzy, a następnie prowadzi na nich obliczenia zgodnie z algorytmem SMO, który został opisany w rozdziale 5.2.5. Podczas obliczeń przeprowadzonych jest bardzo wiele czasochłonnych operacji, jak np. mnożenie macierzy. Obliczenia te wykonywane są równolegle korzystając z wielu rdzeni procesora w przypadku implementacji na CPU, bądź korzystając z karty graficznej w przypadku implementacji sprzętowej. Wynikiem tej części programu jest zbiór współczynników α oraz parametrów b, które definiują wektory wspierające dla jednej klasy. W celu wytrenowania klasyfikatora wieloklasowego, obliczenia są powtarzane N-razy, gdzie N oznacza liczbę klas - 8 w przypadku zbioru R-8, 4 jeśli chodzi o zbiór WebPK. W związku z tym finałowym rezultatem omawianego modułu, jest N-elementowy zbiór wektorów wspierających, dzięki któremu możliwe jest sklasyfikowanie dokumentu do konkretnej klasy.

IV. **Klasyfikacja** - w module tym realizowana jest klasyfikacja danych treningowych oraz testowych przy użyciu formuły (5.67).

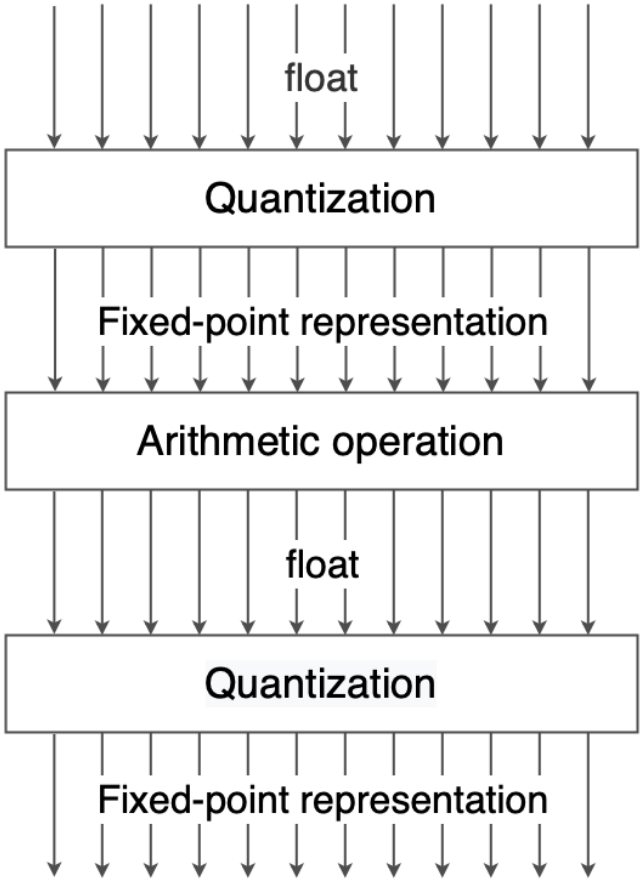
5.4.2. Implementacja w procesorach ogólnego przeznaczenia

Implementacja treningu wektorów nośnych w CPU została wykonana w języku C++, przy użyciu biblioteki OpenMP [100], dzięki której możliwe jest przeprowadzenie obliczeń w sposób równoległy. Dodatkowo w celu poprawienia efektywności obliczeń użyto specjalnej dyrektywy kompilatora *-03*, której użycie optymalizuje implementacje na wielu rdzeniach. Eksperyment został uruchomiony w procesorze Intel(R) Xeon(R) CPU E5-2630 v3(2.4GHz). Głównym celem opisywanej implementacji jest zbadanie jaki wpływ na efektywność algorytmu ma użycie zredukowanej precyzji danych w obliczeniach w trakcie treningu SVM. Należy nadmienić, że obliczenia zostały prowadzone przy użyciu typu *float*, gdyż w

procesorach ogólnego przeznaczenia nie jest możliwe użycie danych ze zdeklarowaną przez użytkownika szerokością bitową taką jak np. 13 bitów. Jednak używając odpowiednich metod kwantyzacji możemy zredukować informację przechowywaną w zmiennej 32-bitowej do pożądanej liczby bitów. Dzięki czemu możliwe jest sprawdzanie, do jakiej precyzji można zredukować szerokość, by zachować efektywność algorytmu. Zmiana szerokości bitowej z 16-bitów do na przykład 15-bitów w procesorach ogólnego przeznaczenia nie spowoduje żadnych zmian w czasie wykonania algorytmu, gdyż obliczenia będą wykorzystane i tak w tym samym *hardware*, więc czas wykonania dla 15-bitów będzie dokładnie taki sam jak dla 16 bitów. Implementacja w CPU pełni rolę symulacyjną, sprawdzającą skuteczność zaproponowanych metod kwantyzacji. Zdobyta w ten sposób informacja może zostać użyta do implementacji algorytmu w platformach akceleracyjnych, jak na przykład układy **FPGA**, gdzie nie obowiązuje żaden standard deklarujący szerokość bitową w jakiej winny być reprezentowane dane, a co za tym idzie programista ma w tej kwestii pełną dowolność, więc zmiana szerokości bitowej będzie miała duży wpływ na szybkość wykonania algorytmu. Innym typem akceleratorów, w których zredukowana precyzja ma wpływ na czas wykonania są układy **GPUGPU** (rozdział 3.2), dla których implementacja wraz z wynikami została przedstawiona w kolejnej sekcji. W obu przypadkach użycie mniejszej szerokości niesie ze sobą korzyści akceleracyjne oraz pamięciowe. Do zmiany szerokości bitowej liczby z jej oryginalnej 32-bitowej (*floating-point*) do n-bitowej reprezentacji ($n < 32$), zostały użyte dwa algorytmy opisane w rozdziałach 5.3.2 oraz 5.3.3. W celu zapewnienia, iż wszystkie obliczenia będą przeprowadzone używając tej samej *fixed-point* reprezentacji, kwantyzacja jest dokonywana na danych wejściowych do danej funkcji oraz na jej rezultacie, który jest jednocześnie wejściem do kolejnych obliczeń. W przypadku, gdy wejściem do obliczeń jest macierz lub wektor, kwantyzacja realizowana jest równolegle. Proces ten ilustruje rysunek 5.9. Otrzymane wektory wspierające są użyte do klasyfikacji, w której dane wejściowe również są kwantyzowane do tej samej szerokości w jakiej odbył się trening. Wyniki opisanej implementacji dla dwóch zaproponowanych metod kwantyzacji rozdziały 5.3.2 oraz 5.3.3, zostały przestawione w tabelach 5.1 i 5.2. W obu przypadkach znaczący spadek skuteczności algorytmu odnotowano dla szerokości mniejszej niż 6 bitów. W przypadku metody *Max magnitude dynamic fixed-point quantization* (rozdział 5.3.2) jest to spowodowane bardzo dużą ilością zer, które są wynikiem tej kwantyzacji dla reprezentacji 4-bitów i 5-bitów. W związku z czym algorytm ma trudności z poprawnym sklasyfikowaniem tekstów, gdyż w wielu przypadkach wartości dla różnych klas są podobne. Natomiast w metodzie *Min-max dynamic fixed-point quantization* (rozdział 5.3.3), efekt ten powoduje zbyt mała liczba wyprodukowanych kubełków, dla reprezentacji 4 i 5 bitowej. W wyniku czego kwantyzacja zwraca te same wartości dla zbyt dużego zakresu liczb, więc podobnie jak w przypadku zbyt dużej liczby zer, powstaje problem z dokonaniem poprawnego rozróżnienia dokumentów.

Tabela 5.1. Skuteczność SVM po zastosowaniu kwantyzacji typu *max magnitude dynamic fixed-point*

Zbiór danych	**Skuteczność [%] dla wybranej reprezentacji [bity]**						
	32	**16**	**8**	**7**	**6**	**5**	**4**
Reuters r-8	95.59	95,59	95,03	94,81	94,97	91,9	90,43
WebKG	82,82	82,77	81,94	81,21	81,32	73,25	70,34



Rysunek 5.9. Kwantyzacja podczas treningu SVM

5.4.3. Implementacja w układach GPGPU

Obliczanie wektorów SVs w układach GPGPU (rozdział 3.2) zostało zrealizowane w środowisku CUDA [110] (rozdział 3.2.2) przy pomocy biblioteki CUBLAS [97], która dostarcza optymalnych implementacji podstawowych operacji z algebry linowej. Obliczenia zostały uruchomione na karcie Nvidia Tesla V100-SXM2-32GB [96] (rozdział 3.2.1). posiadającej rdzenie *Tenosr cores*, które zostały opisane w rozdziale 3.2. Jednym z warunków skorzystania z *tenos cores* jest użycie jednego z następujących formatów danych $half, int\_8$ lub int_4. Zaproponowane w niniejszej pracy metody kwantyzacji mogą być użyte do konwersji do typu *half*, który do reprezentacji liczby używa 16 bitów. Głównym celem implementacji na GPU, jest pokazanie wpływu użycia zredukowanej precyzji na czas wykonania obliczeń, jak również na niezbędną ilość pamięci potrzebną do zrealizowania procesu uczenia. Jako wejściowe typy danych zastosowano *double-precision*, *single-precision* oraz wspomniany wcześniej *half-precsion*. Obliczenia na danych typu *half-precision* przeprowadzano przy użyciu biblioteki Cuda-Math-Api [98]. Zawiera ona szereg funkcji matematycznych oraz udostępnia funkcje potrzebne do konwersji między typami. Karty graficzne w bardzo efektywny sposób dokonują obliczeń takich jak:

– możenie macierz-wektor,

– mnożenie wektor-wektor,

Tabela 5.2. Skuteczność SVM po zastosowaniu kwantyzacji typu *min-max dynamic fixed-point*

Zbiór danych	**Skuteczność [%] dla wybranej reprezentacji [bity]**						
	32	**16**	**8**	**7**	**6**	**5**	**4**
Reuters r-8	95.59	95,02	94,82	94,67	93,94	92,99	92,05
WebKG	82,82	82,79	81,81	81,46	81,65	74,91	70,16

– iloczyn skalarny dwóch wektorów.

Operacje te stanowią serce wszystkich obliczeń koniecznych do przeprowadzenia treningu SVM i w opisywanej implementacji są uruchamiane właśnie w układach GPGPU. Dodatkowo wymienione operację są używane przez większość algorytmów uczenia głębokiego. Mając odpowiednie algorytmy konwertujące dane z ich 64 bądź 32 bitowych do 16 bitowej reprezentacji oraz mając pewność, że przy ich użyciu algorytm zachowa swoją skuteczność, można w znaczący sposób dokonać akceleracji długotrwałych procesów treningu. Dodatkową intencją opisywanej implementacji było zaprezentowanie na przykładzie SVM, a dokładniej posługując się rozmiarami danych będącymi wejściem do SVM, na ile możliwe jest poprzez użycie zredukowanej szerokości przyśpieszenie operacji cząstkowych. Prócz operacji macierzowo-wektorowych, podczas trenowania wektorów nośnych, konieczne jest dokonanie obliczeń na pojedynczych wartościach jak np. obliczenie parametru b. Obliczenia tego typu są realizowane w procesorze CPU. Jako zbiór wejściowy użyto Reuters Dataset-r8 (rozdział 5.4). Po zastosowaniu konwersji TF-IDF (rozdział 5.1.2) na podanym zbiorze, otrzymano macierz o rozmiarach (2538, 827), przy użyciu której zmierzono czas potrzebny na wykonanie trzech wyżej wymienionych, najbardziej czasochłonnych obliczeń, koniecznych do przeprowadzania treningu SVM . Wyniki tych pomiarów, dla trzech typów danych, wraz z uwzględnieniem czasu potrzebnego na skopiowanie danych wejściowych z procesora na kartę jak również rezultatów z karty na procesor, zostały zawarte w tabeli 5.3. Jak można zauważyć, używając typu *half* uzyskano prawie trzykrotne przyśpieszenie, na najbardziej czasochłonnej operacji czyli przemnożeniu wektora przez macierz, co jest bardzo znaczącym przyśpieszeniem. Na pozostałych dwóch operacjach uzyskano przyśpieszenie na poziomie 26-27%, co również jest bardzo dobrym wynikiem. Pokazuje to jak bardzo istotnym czynnikiem podczas uruchamiana treningu jest zastosowanie zredukowanej precyzji. Czas potrzebny na obliczenie skalara dwóch wektorów, jest porównywalny do czasu koniecznego na wykonanie przemnożenia wektora przez wektor, mimo iż jest to operacja bardziej złożona obliczeniowo. Wynika to z faktu, że rezultatem skalara jest jedna liczba, więc występuję tu spora oszczędność czasu, koniecznego na skopiowanie wyników z karty w procesor.

Tabela 5.3. GPGPU - pomiary czasów operacji macierzowych [ms]

Typ danych	**Wektor*Wektor**	**Macierz*Wektor**	**Iloczyn skalarny**
double	0,66	3,04	0,64
float	0,59	1,95	0,58
half	0,52	1,12	0,51

Wykonując cały proces uczenia wektorów SVs, używając danych 16-sto bitowych, względem pojedynczej oraz podwójnej precyzji dla obliczenia pojedynczej klasy wektorów, uzyskano przyśpieszenia kolejno **1.26** oraz **1.76** razy. Dla zastosowanego zbioru danych czas potrzebny na obliczenie wektorów dla jednej klasy, używając typu *double*, wynosi ok. półtorej godziny. Oczywiście czas ten może się nieco różnić dla każdej z klas, niemniej jednak nie są to duże różnice, i tak dla ośmiu klas w przypadku zbioru r-8, algorytm potrzebował ok. 11 godzin na dostarczenie ośmiu zestawu wektorów klasyfikujących. Używając typu *half* czas ten zmalał do 6,5 godziny, więc oszczędność jest zauważalna. Kopiując macierz typu *double*, zbudowaną ze zbioru r-8, na kartę graficzną, należy zarezerwować **16,8 [MB]** pamięci, używając do tego celu typu *float*, ilość koniecznej pamięci jest mniejsza dwa razy, natomiast dla *half* okupowanie pamięci spada cztery razy. Spadek wymaganej pamięci odgrywa ogromną rolę przy programwaniu układów GPGPU. Często zdarza się, że wejściowe zbiory są za duże by pomieścić je na karcie, więc należy wykonać sporo wysiłku, aby liczyć algorytm na ich fragmentach po czym łączyć rezultaty przy użyciu procesora. Używając zredukowanej precyzji, wymagana pamięć zostaje znacznie zredukowana, co może doprowadzić do zmniejszenia czasu koniecznego na zaprogramowanie danego algorytmu. Należy podkreślić, że dokonując konwersji udostępnionej przez Cuda-Math-Api dla całego procesu SVM, spadek skuteczności dla 16-bitów był bardzo zbliżony do tego, który został uzyskany przez zaproponowane w tej pracy metody kwantyzujące. Dodatkową korzyścią akceleracyjną oraz pamięciową byłoby użycie jako typu *int_8*, jednak w tym przypadku należy użyć metody kwantyzującej dane z typu zmienno-przecinkowego to typu całkowitego. Rozwiązanie takie jest używane w popularnej bibliotece *TensorFlow* [101], i jest brane pod uwagę jako przyszłe zadanie badawcze.

6. Efektywne sposoby obliczania operacji konwolucji przy użyciu układów GPGPU

Niniejszy rozdział opisuje efektywne sposoby przetwarzania warstw konwolucyjnych (ang. *convolutional layers*), do których teoria została przedstawiona w rozdziale 2.2.1. Obecne architektury splotowych sieci neuronowych składają się z kilkunastu warstw konowlucyjnych, których obliczenie zajmuje większość czasu, oraz kilku warstw w pełni połączonych (*fully connected layers*). Z tego też względu akceleracja warstw konwolucyjnych jest tematem niniejszego rozdziału. Przeprowadzenie procesu uczenia tak obszernej architektury, zawierającej miliony parametrów, wymaga wielu dni, wykorzystując przy tym wiele kart graficznych, które obecnie są najbardziej efektywnym akceleratorem do wykonania tego typu obliczeń. Cały proces wymaga zatem zużycia ogromnej liczby energii. W związku z tym niezwykle istotna jest kwestia akceleracji przetwarzania warstw konwolucyjnych, stąd też na przestrzeni ostatnich lat opracowywano nowe sposoby prowadzące do zredukowania czasu koniecznego do przeprowadzenie obliczeń. W niniejszym rozdziale zostaną omówione najbardziej efektywne algorytmy obliczające konwolucję, które wraz z ich powstawaniem stawały się częścią biblioteki cuDnn (The Nvidia deep neutral network library) [99]. Biblioteka cuDnn dostarcza najbardziej efektywnych algorytmów służących do obliczeń związanych z uczeniem maszynowym w układach GPGPU, w tym omawianych warstw konwolucyjnych. W zależności od liczby próbek wejściowych, wielkości pojedynczej z nich, rozmiaru filtru oraz typu danych, cuDnn wybiera algorytm który w takich warunkach obliczy konwolucje w sposób najbardziej efektywny. Z drugiej jednak strony w filtrach warstw konwolucyjnych znajduje się wiele wag, których wartość jest równa zeru, co jest wynikiem operacji *Pruningu* [81], który jako istotna część opisywanego rozdziału zostanie w jednym z kolejnych podrozdziałów bliżej przedstawiony. Fakt występowania dużej ilości zer stał się motywacją do próby akceleracji przetwarzania warstw konowucyjnych przy użyciu operacji związanych z macierzami rzadkimi. Spsób ten będzie dalej nazywany *konwolucją rzadką*. W rozdziale tym główna uwaga została poświęcona na zbadanie, kiedy warto użyć sposobu przetwarzania macierzy rzadkich do obliczenia konwolucji w miejsce biblioteki cuDnn. Dodatkowo sprawdzany jest wpływ użycia zredukowanej precyzji (w tym przypadku typu *half*) na szybkość obliczeń zarówno dla konwolucji rzadkiej oraz dedykowanej biblioteki cuDnn. Jako punkt odniesienia zostały użyte wybrane najnowsze modele, których architektury oraz wykorzystanie zostaną dokładniej opisane. W celu ujednolicenia oznaczeń posłużono się nazwami parametrów zaproponowanymi w rozdziale 2.2.1, gdzie:

– N - rozmiar batcha,

– K - liczba filtrów (kanałów wyjściowych),

– C - liczba kanałów wejściowych (głębia),

– H/W - wysokość/szerokość wejścia,

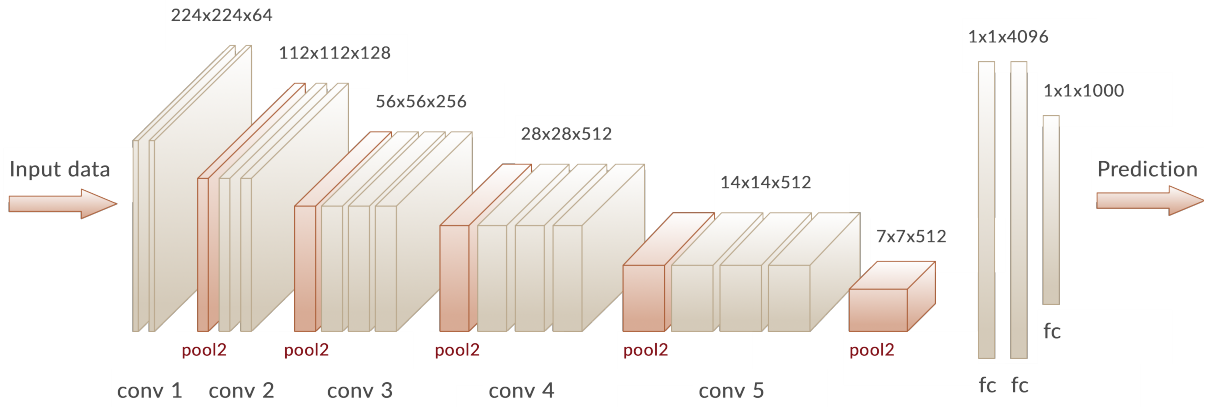
– R/S - wysokość/szerokość filtru,

– E/F - wysokość/szerokość wyjścia.

6.1. Architektury wybranych modeli konwolucyjnych

6.1.1. Architektura sieci VGG-16

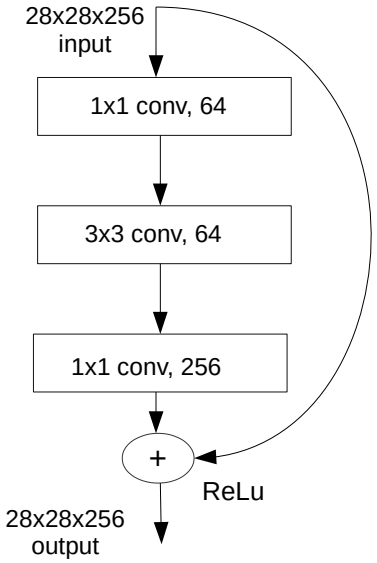
Model VGG-16 [94] jest konowolucyjną siecią neuronową która wygrała międzynarodowe zawody ILSVR (Imagenet)[105] w 2014 roku. Jednocześnie pozostaje jako jedna z najbardziej efektywnych architektur po dzień dzisiejszy. Z tego też względu została ona użyta na potrzeby niniejszej pracy, jako przykład modelu, na którym będą badane algorytmy przetwarzające operację konwolucji. Charakterystyczne dla modelu VGG-16 jest posiadanie filtrów dla każdej warstwy konwolucji o rozmiarach 3×3, przy parametrze *stride* ustawionym na 1 oraz jednakowych warstw typu *pool*, które używają funkcji *max* z filtrami o rozmiarach 2×2 z parametrem *stride* równym 2. Za warstwami konowucyjnymi znajdują się 3 warstwy w pełni połączone, zakończone funkcją *Softmax* dokonującą predykcji. Wejściowe obrazy zapisane w formacie RGB posiadają rozmiary $244 \times 224 \times 3$. Jest to bardzo obszerna architektura, która zawiera w przybliżeniu 138 milionów parametrów. Ułożenie kolejnych warstw wraz ze zmieniającymi się rozmiarami danych, zostało przedstawione na rysunku 6.1. Model VGG-16 w przedstawianych w tej pracy eksperymentach służy do zbadania możliwości przyśpieszenia obliczenia operacji konwolucji typu 2D.



Rysunek 6.1. Architektura VGG-16

6.1.2. Konwolucja typu 1×1 pochodząca z sieci ResNet

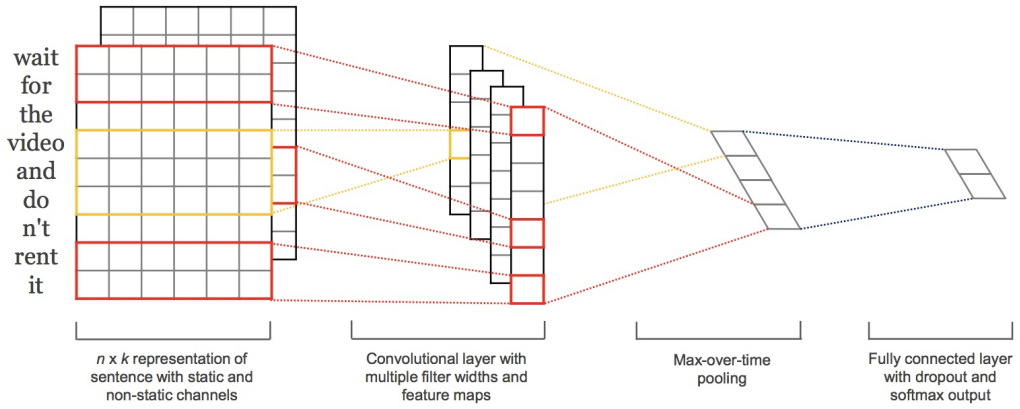
Jak zostało powiedziane w rozdziale 2.2.1 konwolucja z filtrem o rozmiarach 1×1 jest wykorzystywana do zmiany rozmiaru sieci, a dokładniej mówiąc do manipulowania wartością parametru *głębia* (rozdział 2.2.1). W praktyce zabieg taki stosowany jest przed kosztowną konwolucją, taką jak na przykład 3×3 bądź 5×5. Przykładem redukowania rozmiarów danych za pomocą omawianej konwoucji jest architektura ResNet [41], gdzie jest ona użyta przed obliczeniem splotu 3×3, gdzie redukuje *głębię* z 256 do 64, a następnie ten sam typ konwoujcji jest użyty do powrotu do oryginalnej wartości tego parametru po dokonanych obliczeniach. Operacja ta została przedstawiona na rysunku 6.2. Obie warstwy konwolucji 1×1 zostały poddanie próbie akceleracji przy pomocy operacji na macierzach rzadkich.



Rysunek 6.2. Użycie konwolucji 1x1 w modelu ResNet

6.1.3. Architektura CNN-non static

CNN-non static [50] jest konwolucyjną siecią neuronową używaną do przetwarzania języka naturalnego (rozdział 5.1), a dokładniej do klasyfikacji tekstu. Ze względu na specyfikację danych tekstowych jest to konwoulcja typu 1D, której architektura posłużyła do eksperymentów akceleracyjnych w niniejszej pracy. Konwoulcuja 1D jest bardzo efektywna przy ekstrakcji cech z segmentów danych o ustalonej długości z całego zbioru, gdy położenie cechy w danym segmencie nie odgrywa ważnej roli. Model składa się z dwóch warstw, działających równolegle, zawierających po 128 filtrów, których rozmiary wynoszą odpowiednio 2×300 oraz 3×300. Wyniki obu tych warstw są łączone w warstwę gęstą posiadającą 128 wyjść, po których znajduję się funkcja *Softmax*. Jako wejście służą przetrenowane wektory o rozmiarach 64×300, będące produktem algorytmu *GloVe* opisanego dokładnie w rozdziale 5.1.3.2. Działanie sieci dobrze obrazuje rysunek 6.3 pochodzący z [50].



Rysunek 6.3. Architektura CNN-non static [50]

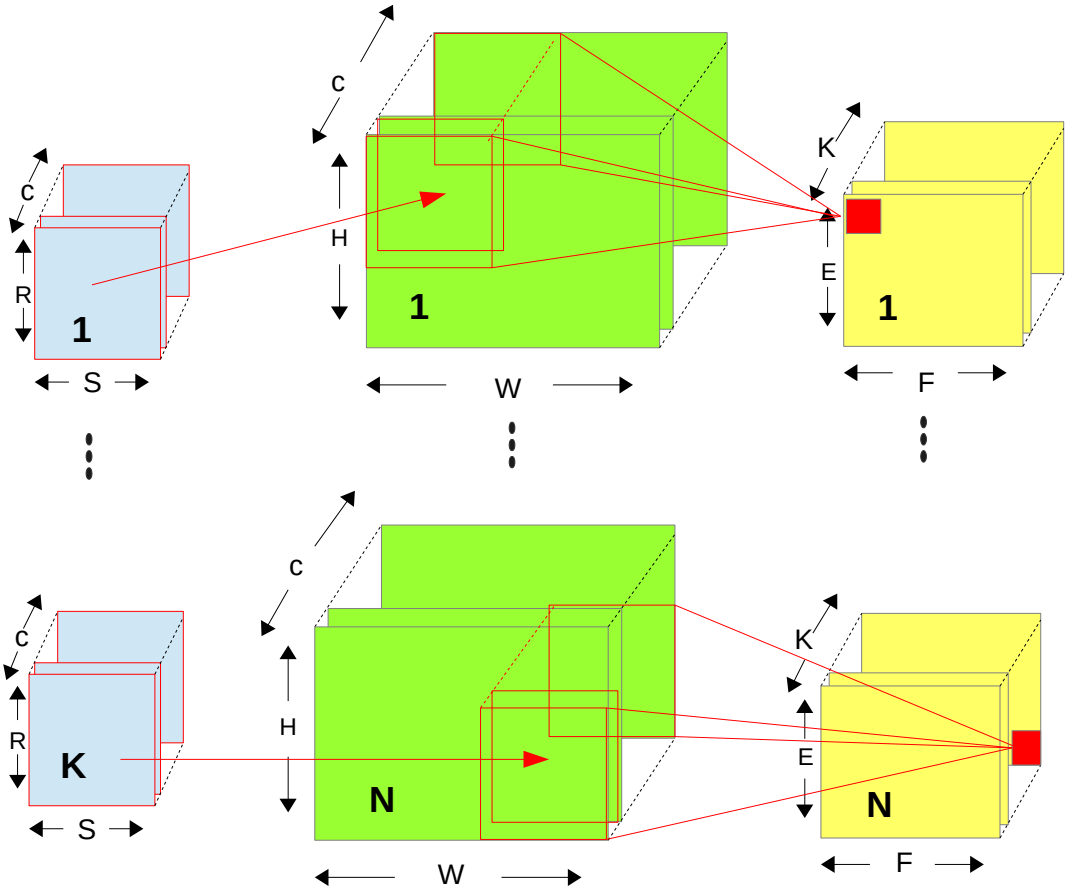
6.2. Efektywne sposoby liczenia konowolucji w układach GPGPU

Sekcja ta opisuje efektywne sposoby obliczenia konwolucji z punktu widzenia kart graficznych. Opisane metody stanowią część dedykowanej biblioteki cuDnn, która dla danego rozmiaru konwolucji wybiera najbardziej efektywny z nich. Dlatego też w celu sprawdzenia, który algorytm został wybrany, jak również jego szybkości, w ustawieniach cuDnn została wybrana flaga **CUDNN_CONVOLUTION_FWD_PREFER_FASTEST**. Wyniki czasowe jak i użyty algorytm do przetworzenia danej knowolucji zostały zawarte w sekcji 6.5. Użyta biblioteka daje możliwość ręcznego wskazania algorytmu, jednak jak pokazały eksperymenty, pozostawienie wyboru bibliotece daje najlepsze efekty. Karty graficzne są bardzo efektywne w obliczaniu obszernych działań macierzowych takich jak mnożenie macierzy czy obliczanie skalaru. Z tego względu opisane metody sprowadzają się do ułożenia w ten sposób danych wejściowych, aby możliwe było przeprowadzenie konowlucji właśnie za pomocą tych operacji. Oczywiste jest, że po dokonanych obliczeniach będzie wymagana jeszcze jedna transformacja, ale jak się okazuje szybkość przetworzenia samej konwolucji przy pomocy wspomnianych operacji macierzowych, niweluje starty poniesione przy dokonywaniu obu transformacji.

6.2.1. Obliczanie konowlucji metodą bezpośrednią

Najprostszym, a zarazem najmniej efektywnym sposobem na przeprowadzenie obliczeń konwolucji z punktu widzenia układów GPGPU, jest metoda bezpośrednia (ang. *direct convolution*), gdzie wartości kernela przemnażane są z odpowiadającymi im wartościami z danych wejściowych. Następnie filtr przesuwany jest zgodnie z wartością parametru *stride* (*krok*), po czym następuje kolejne mnożenie oraz sumowanie z wynikami poprzedniego mnożenia. Zebrane w ten sposób cząstkowe wyniki w postaci macierzy o rozmiarach wyjścia sumowane są z zebranymi w ten sam sposób wynikami zebranymi wzdłuż parametru *depth* (*głębia*). Aby obliczyć konwolucję wejścia o rozmiarze *n* z filtrem o tym samym rozmiarze wymagane jest wykonanie n^2 mnożeń oraz *n(n-1)* sumowań co finalnie daje kwadratową złożoność obliczeniową. Przebieg obliczenia konwulucji metodą bezpośrednią został przedstawiony zarówno

za pomocą poniższego pseudokodu oraz na ilustracji 6.4. Jak pokazały eksperymenty metoda ta, chociaż jest częścią biblioteki cuDnn, nie została wykorzystywana do przetworzenia badanych w tej pracy warstw konwolucyjnych.



Rysunek 6.4. Bezpośrednie obliczanie konwolucji

6.2.2. Obliczanie konowlucji przy użyciu mnożenia macierzy

Efektywnym sposobem do liczenia konwolucji w układach GPGPU jest użycie możenia macierzy. Efektywność tego sposobu bierze się z faktu dobrego przysposobienia kart do tego typu operacji [18]. Aby było możliwe takie obliczenie konowulucji, należy stworzyć dwie macierze, poprzez odpowiednie poukładanie zarówno danych wejściowych jak i wag. Wymiary macierzy tworzonej na podstawie danych wejściowych, nazywanej dalej macierzą *D*, są ustalane na podstawie wielkości danych wejściowych oraz filtru z uwzględnieniem wartości parametrów *stride*, *padding* i *depth*. W efekcie macierz *D* ma rozmiary $CRS \times EF$ co jest równoważne z zapisem: $[new\_high * new\_width \times kernel\_high * kernel\_width * depth]$, gdzie parametry new_high oraz new_width obliczane są za pomocą formuły:

$$\begin{aligned} new\_high &= \frac{input\_high - kernel\_high + 2 * padding}{stride} + 1 \\ new\_width &= \frac{input\_width - kernel\_width + 2 * padding}{stride} + 1 \end{aligned} \tag{6.1}$$

Algorytm 11. Algorytm sekwencyjnej metody bezpośredniej konowlucji

```
1: function CONV(input, weighs, output)
2:     for n := 0 to N − 1 do
3:         for k := 0 to K − 1 do
4:             for c := 0 to C − 1 do
5:                 for h := 0 to E − 1 do
6:                     for w := 0 to F − 1 do
7:                         for r := 0 to R − 1 do
8:                             for s := 0 to S − 1 do
9:                                 out[n][m][h][w] += input[n][c][h+r][w+s]*weights[k][c[r][s]
   end function
```

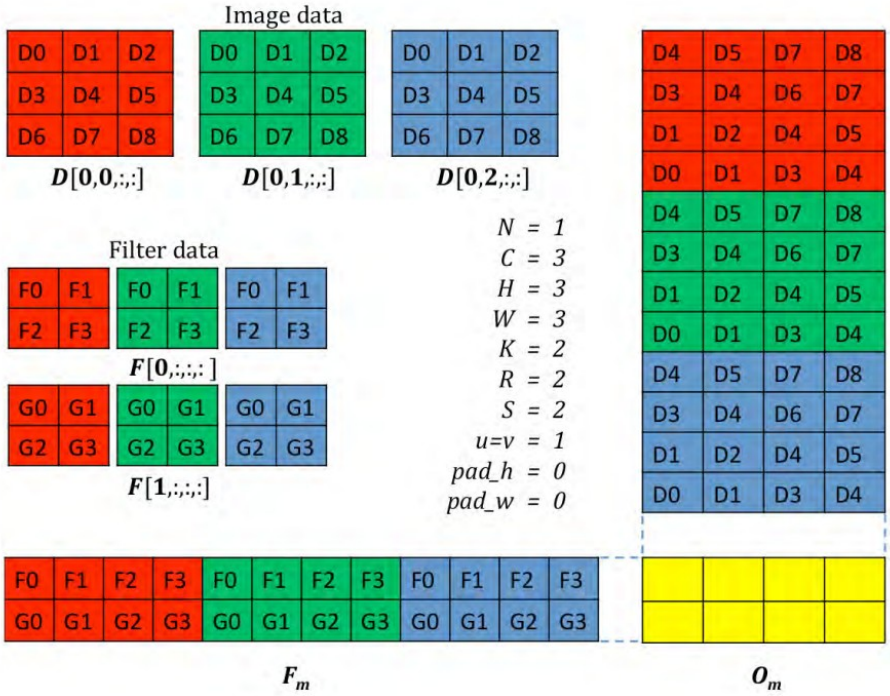
Ostatecznie powstała macierz zawiera $R \times S$ więcej danych niż w postaci oryginalnej, które są duplikatami danych oryginalnych, co pociąga za sobą większe wymogi pamięciowe. Druga macierz jest budowana z filtrów, poprzez zmianę ich rozmiarów na $K \times CRS$ i jest nazywana dalej macierzą *F*. Należy zauważyć, że w tym przypadku nie są duplikowane dane, a macierz budowana jest z wykorzystaniem kanałów wyjściowych. Dane w nowej macierzy wejściowej układane są w taki sposób, by możliwe było obliczenie konwolucji jako produktu skalarnego odpowiedniego rzędu i kolumny. Operacja ta jest wykonywana dla każdego rzędu transformowanej macierzy wejściowej i każdej kolumny transformowanej macierzy wag. Algorytm w jaki transformowane są obie macierze najlepiej obrazuje bardzo często cytowana ilustracja 6.5, pochodząca z [11], gdzie omawiana transformacja została pokazana dla pierwszej warstwy czyli dla liczby kanałów wejściowych równych 3. Dla późniejszych warstw algorytm wygląda tak samo, jednak zwykle zwiększa się liczba kanałów, co uniemożliwia poprawne zobrazowanie sytuacji, stąd też autorzy w przykładzie posłużyli się filtrem o rozmiarze 2. Jako rezultat powstaje macierz *O* o wymiarach $K \times EF$, która wymaga prostej transformacji do rozmiarów $E \times F \times K$, czyli takiej jaka jest wynikiem operacji konwoulcji. Jak się eksperymentalnie okazało (rozdział 6.5) sposób ten używany jest przez bibliotekę cuDnn, w przypadku konowlucji 1D, konwolucji typu 1×1 oraz w momencie, gdy parametr *głębia* nie jest zbyt duży, co ma miejsce zwykle dla pierwszej warstwy, gdy wynosi on 3 (RGB).

6.2.3. Obliczanie konowlucji przy użyciu szybkiej transformacji Fouriera

Szybka transformacja Fouriera (FFT) może być użyta do obliczenia konwolucji, gdyż zgodnie z definicją mówiącą, że *splot dwóch sygnałów wejściowych w dziedzinie czasu jest równoważny z mnożeniem ich transformat Fouriera w dziedzinie częstotliwości* [67]. Oznaczając transformację Fouriera jako $\mathcal{F}$, operację splotu jako $*$, natomiast $\bullet$ oznacza *elemnt-wise multpilicaiton* dwóch macierzy, wówczas konwolucja dwóch wejściowych funkcji f oraz g może być wyrażona jako:

$$f * g = \mathcal{F}^{-1}(\mathcal{F}(f) \bullet \mathcal{F}(g)) \tag{6.2}$$

Odwrotna transformacja Fouriera oznaczona jako $\mathcal{F}^{-1}$ jest wykonywana w celu powrócenia do pierwotnej dziedziny. Pomnożenie transformat sygnałów w ich oryginalnej formie doprowadziłoby do policzenia splotu kołowego (ang. *circuarl convolution*), czyli przeciwnego do tego wyrażonego równaniem 2.23



Rysunek 6.5. Transformacja danych w celu obliczenia konwolucji przy pomocy mnożenia macierzy [11]

[67], co wprowadziłoby efekt zniekształcenia, ponieważ próbki sygnału wejściowego umiejscowione na jego krawędziach zostałyby dodane do siebie, co doprowadziłoby do błędnego w tych miejscach obliczenia splotu [1][67]. Aby temu zapobiec, zarówno dane wejściowe jak i filtry na krawędziach, są uzupełniane zerami (*zero padding*), tak aby finalnie oba miały takie same rozmiary oznaczane jako $L + M - 1$, gdzie L i M oznaczają długości splatanych sygnałów. Metoda ta nazywa jest *szybką konwolucją*, gdyż jej złożoność obliczeniowa wynosi: $O(nlog_2(n))$. Z punktu widzenia układów GPGPU, operacja typu *Hadamard product* (inaczej element-wise multiplication), jest bardzo efektywna co rekompensuje straty poniesione na dokonanie wspomnianych transformat, wymaganych do poprawnego przeprowadzenia obliczeń. W rzeczywistości stosuje się metody *overlap-and-save* (OLS) oraz *overlap-and-add* (OLA) [85], które rozdzielają próbki wejściowe na mniejsze segmenty o ustalonej długości, które następnie mogą być przetwarzane niezależnie co daje podstawę, do ich przetwarzania równoległego, co z punktu widzenia kart graficznych jest ogromną korzyścią. Obie metody różnią się od siebie sposobem radzenia sobie z nadmiarowymi próbkami splotu. OLS dokonuje zakładkowania (ang. *overlap*) sygnału przed dokonaniem splotu i usunięciu przed zapisem z każdego segmentu danych nadmiarowych i zapisaniu tylko poprawnych z nich na odpowiadających im miejscach w macierzy wyjściowej. Natomiast OLA dodaje do siebie nadmiarowe próbki z sąsiadujących bloków w celu uzyskania poprawnych wyników. Dokładniejsze opisy obu metod wraz z próbami ich akceleracji można znaleźć w [1], nie są one częścią niniejszej pracy, gdzie do porównania zaproponowanej metody obliczania konwolucji użyto gotowych rozwiązań z biblioteki cuDnn.

Podsumowując, obliczenie *szybkiej konwoulcji* dla każdego segmentu wymaga czterech kroków:

1. zamianie z dziedziny czasowej na dziedzinę częstotliwości wraz z wymaganym *zero padding*, tak

by rozmiar danych wejściowych i filtru były takie same (dla filtrów jest to dokonywane tylko raz i może być zrobione w tzw. pre-procesingu),

2. dokonaniu mnożenia typu *element-wise* filtru oraz wejściowego segmentu danych,

3. obliczeniu odwrotnej transformacji Fouriera, aby powrócić z powrotem do dziedziny czasowej,

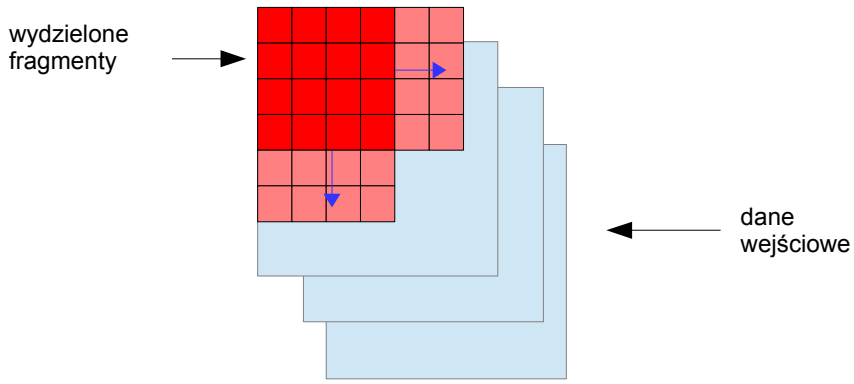
4. odrzuceniu nadmiarowych próbek znajdujących się na krawędziach, powstałych w wyniku *zero paddingu*.

Metoda *szybkiej konowlucji* wykorzystywana jest dla dużych rozmiarów filtrów. Bazując na eksperymentach przeprowadzonych na potrzeby tej pracy, cuDnn skorzystał z tej metody podczas przetwarzania architektury VGG-16 (rozdział 6.1.1), gdy wielkość danych wejściowych była równa bądź mniejsza 58×58 (czyli od 6 warstwy) oraz co ciekawe, gdy ilość próbek w partii danych była większa od 32.

6.2.4. Obliczanie konwolucji przy pomocy algorytmu Winograd

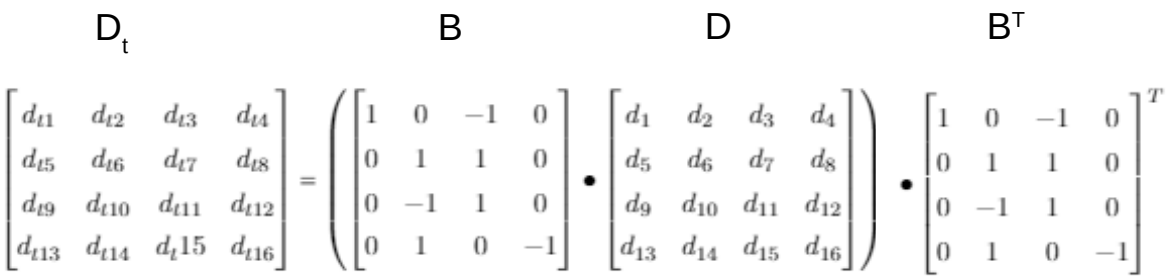
Winograd algorytm [61] nadaje się do obliczenia konwolucji, gdy rozmiar filtrów jest mały, przez co należy rozumieć na przykład filtry o rozmiarze 3×3 i na przykładzie takich filtrów zostanie ta metod przybliżona. Dla takiego rozmiaru filtru, aby możliwe było obliczenie konwolucji przy pomocy opisywanej metody, wejściowe dane muszą być rozmiarów większych niż 4. Mając dane spełniające te warunki, oznaczając fragment macierzy wejściową oraz filtry kolejno jako D oraz F, natomiast macierz wyjściową jako O, należy wykonać następujące kroki:

1. W pierwszym kroku algorytmu dane wejściowe dzielone są na mniejsze kawałki o rozmiarach 4×4 z krokiem równym 2, co jest powtarzane dla każdego kanału wejściowego z osobna. W efekcie czego otrzymane fragmenty mają rozmiary $C \times 4 \times 4$ (C oznacza głębię). Krok ten przedstawia rysunek 6.6.



Rysunek 6.6. Algorytm Winograd - podział danych wejściowych na mniejsze fragmenty

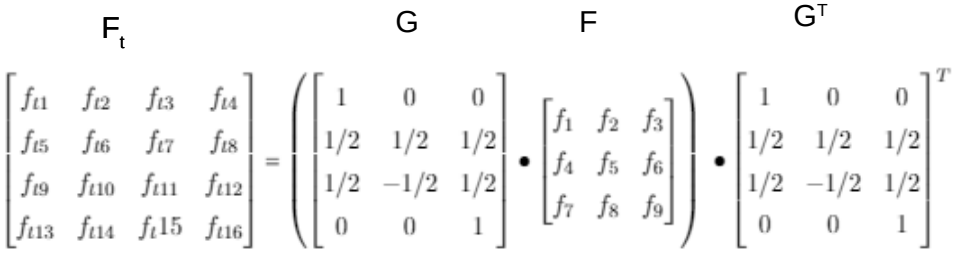
2. W drugim kroku każdy z uprzednio powstałych fragmentów jest transponowany poprzez przemnożenie go przez specjalną macierz *B*, zawierającą wartości ze zbioru {-1, 0 ,1}. Konstrukcja macierzy dokonującej transformacji pozwala na skonstruowanie *minimal fitering alghorighm*, który oparty jest na *Chinese remaider theory (CRT)*, która dokładnie jest opisana w [123]. Format macierzy transformującej B, wraz z całą transformacją, została zilustrowana na rysunku 6.7.



$$\begin{bmatrix} d_{t1} & d_{t2} & d_{t3} & d_{t4} \\ d_{t5} & d_{t6} & d_{t7} & d_{t8} \\ d_{t9} & d_{t10} & d_{t11} & d_{t12} \\ d_{t13} & d_{t14} & d_t15 & d_{t16} \end{bmatrix} = \left(\begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \bullet \begin{bmatrix} d_1 & d_2 & d_3 & d_4 \\ d_5 & d_6 & d_7 & d_8 \\ d_9 & d_{10} & d_{11} & d_{12} \\ d_{13} & d_{14} & d_{15} & d_{16} \end{bmatrix} \right) \bullet \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}^T$$

Rysunek 6.7. Algorytm Winograd - transformacja wejściowych fragmentów

3. W trzecim kroku dokonywana jest transformacja podobna do tej z kroku poprzedniego, jednak w tym przypadku transformowane są filtry, poprzez przemnożenie przez macierz *G*, której budowa ma podstawy teoretyczne również w CRT [123]. Zawartość macierzy *G* oraz opisywana transformacja zostały przedstawione na ilustracji 6.8.



$$\begin{bmatrix} f_{t1} & f_{t2} & f_{t3} & f_{t4} \\ f_{t5} & f_{t6} & f_{t7} & f_{t8} \\ f_{t9} & f_{t10} & f_{t11} & f_{t12} \\ f_{t13} & f_{t14} & f_t15 & f_{t16} \end{bmatrix} = \left(\begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} f_1 & f_2 & f_3 \\ f_4 & f_5 & f_6 \\ f_7 & f_8 & f_9 \end{bmatrix} \right) \bullet \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix}^T$$

Rysunek 6.8. Algorytm Winograd - transformacja filtrów

4. W czwartym kroku dokonywana jest operacja *element-wise multiplication* (oznaczona jako $\odot$), na powstałych w wyniku wcześniejszych transformat macierzach, co obrazuje rysunek 6.9.

5. Ostatnim piątym krokiem jest transformacja macierzy wynikowej *O* do macierzy o rozmiarach 2×2, która zarazem jest finalnym wynikiem konwolucji. Transformacji dokonuje się poprzez macierz *A*, która również wywodzi się z teorii CRT [123], której wartości jak również całą finałową transformację opisuje rysunek 6.10.

Podsumowując, operację obliczenia konwolucji dla danego fragmentu danych wejściowych, przy użyciu metody Winograd, posługując się wcześniej wprowadzonymi oznaczeniami, można zapisać jako:

$$S = A^T \Big[[GFG^T] \bullet [B^T DB] \Big] A \tag{6.3}$$

$$\underset{O_t}{\begin{bmatrix} o_{t1} & o_{t2} & o_{t3} & o_{t4} \\ o_{t5} & o_{t6} & o_{t7} & o_{t8} \\ o_{t9} & o_{t10} & o_{t11} & o_{t12} \\ o_{t13} & o_{t14} & o_t15 & o_{t16} \end{bmatrix}} = \underset{F_t}{\begin{bmatrix} f_{t1} & f_{t2} & f_{t3} & f_{t4} \\ f_{t5} & f_{t6} & f_{t7} & f_{t8} \\ f_{t9} & f_{t10} & f_{t11} & f_{t12} \\ f_{t13} & f_{t14} & f_{t15} & f_{t16} \end{bmatrix}} \odot \underset{D_t}{\begin{bmatrix} d_{t1} & d_{t2} & d_{t3} & d_{t4} \\ d_{t5} & d_{t6} & d_{t7} & d_{t8} \\ d_{t9} & d_{t10} & d_{t11} & d_{t12} \\ d_{t13} & d_{t14} & d_{t15} & d_{t16} \end{bmatrix}}$$

Rysunek 6.9. Algorytm Winograd - *element wise multipilcation*

$$\underset{O}{\begin{bmatrix} o_1 & o_2 \\ o_3 & o_4 \end{bmatrix}} = \left(\underset{A}{\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}} \bullet \underset{O_t}{\begin{bmatrix} o_{t1} & o_{t2} & o_{t3} & o_{t4} \\ o_{t5} & o_{t6} & o_{t7} & o_{t8} \\ o_{t9} & o_{t10} & o_{t11} & o_{t12} \\ o_{t13} & o_{t14} & o_{t15} & o_{t16} \end{bmatrix}} \right) \bullet \underset{A^T}{\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}^T}$$

Rysunek 6.10. Algorytm Winograd - transformacja wyjściowa

Zebrane w ten sposób wyniki cząstkowe są sumowane wzdłuż parametru C. Główną zaletą metody Winograd jest fakt, że redukuje ona liczbę koniecznych do przetworzenia konowlucji mnożeń o $\frac{p^2}{m^2(p-2)^2}$, gdzie p oznacza wielkość fragmentu (w opisywanym przykładzie wynosi ona 4), jednocześnie zwiększając liczbę sumowań [61][123], które z kolei są operacjami mniej kosztownymi. Jak się okazuje, algorytm ten jest używany przy obliczaniu niektórych warstw splotowych sieci VGG-16, szczególnie dla tych, gdzie wejścia są stosunkowo duże (większe od 58x58).

6.3. Efektywne wykorzystanie operacji *Pruningu* w celu przechowywania wag w postaci macierzy rzadkich

Pruning jest techniką służącą do redukowania wielkości sieci, które mogą zawierać miliony danych, jak opisana wyżej architektura VGG-16. Owo redukowanie polega na usuwaniu wag bądź połączeń, które mają mały wpływ na finalny wynik funkcji aktywacji. W eksperymentach przeprowadzonych na potrzeby niniejszej pracy, wpływ ma jedynie pruning dokonywany na wartościach wag (ang. *weight pruning*) z tego też względu zostanie on bliżej opisany. Wytrenowana sieć neuronowa posiada dla każdej z warstw zbiór wag, z których wiele ma wartości bliskie zeru. Stąd też mają one niewielki wpływ na ostateczny wynik danej warstwy. Usunięcie takich wag polega na przypisaniu im wartości zero. Selekcja wag, na których dokona się pruning, polega na wyznaczeniu wartości progowej t, poniżej której wagi uznane są

za mało wartościowe i będą mieć wartości równe zeru. Selekcja wag odbywa się zwykle w ustalonej liczbie iteracji, gdzie w każdej z nich kolejni kandydaci są wybierani. Próg do jakiego może spadać efektywność sieci jest wcześniej ustalony i z reguły nie powinien on przekraczać 1%. W momencie, gdy pruning spowoduje zbyt duży spadek skuteczności modelu, aplikowany jest tzw. *reverse pruning*, który przywraca oryginalne wartości wcześniej wyzerowanym wagom. Pruning można podzielić na:

– *Structured pruning* - w tym rodzaju pruningu usuwane są całe bloki wag. Najbardziej popularną odmianą jest redukowanie liczby kanałów w filtrze poprzez wycinanie wszystkich wartości z danego kanału, co jest równoznaczne z redukowaniem parametru *głębia*. Dzięki takiemu podejściu można dokonać akceleracji bez większych zmian w kodzie (w cuDnn wystarczy zmienić parametr *input channels*). Redukowanie liczby kanałów w wytrenowanej sieci zwykle prowadzi do znacznego spadku jej wydajności, z tego też względu to podejście powinno być stosowane w fazie treningu, co pozwala na niwelowanie tego niepożądanego efektu [81]. Więcej na temat efektywności i sensowności stosowania omawianej metody można znaleźć w [43][44] [63] [65][81],

– *Unstructured pruning* - zeruje wartości wag bez specyficznej geometrii, takiej jak usuwanie całych bloków czy kanałów, jak miało miejsce w wyżej opisanej metodzie. Ten rodzaj pruningu może być aplikowany zarówno do wag [40][80][81][125] jak i do danych wejściowych [64] [118]. Z punktu widzenia eksperymentów przeprowadzonych w niniejszej pracy tylko *Unstructured pruning* dokonany na wartościach filtrów ma sens, stąd też dalsze rozważania dotyczą tylko tego typu pruningu. Informację o ilości wyzerowanych wag, uzyskane poprzez omawiany typ pruningu, ciężko wykorzystać w celu akceleracji używając dedykowanych bibliotek, stąd też głównym punktem tego rozdziału jest zaimplementowanie metody, dzięki której możliwa stała się akceleracja obliczeń konwolucji poprzez wykorzystanie poziomu *sparsity* osiągniętego poprzez *Unstructured pruning*.

Aplikując pruning do wyżej opisanych modeli (VGG-16, CNN-Non static, 1x1 conv), które zostały użyte jako punkty odniesienia, zapamiętana zostaje informacja o najniższym poziomie *rzadkości*, który został uzyskany w K różnych filtrach. Informacja ta jest wykorzystana do ujednolicenia ilości zer we wszystkich K filtrach, dzięki czemu znany on jest przed uruchomieniem kernela dokonującego obliczeń konwolucji w GPGPU, co redukuje konieczność obliczania go dla każdego bloku wątków z osobna (blok wątków oblicza pojedynczy filtr wyjściowy, rozdział 6.4), co z kolei ma wpływ na szybkość obliczeń, co dokładniej zostało wytłumaczone w kolejnym podrozdziale 6.4. Wspomniane ujednolicenie poziomu *rzadkości* dla wszystkich filtrów w pojedynczej warstwie polega na potraktowaniu pewnych wartości zerowych jako niezerowych, co oznacza, że biorą one udział w obliczeniach, ale nie mają wpływu na ostateczny wynik. Liczba takich zer jest zależna od różnicy między ilością zer w danym filtrze, a ilością tych wartości w filtrze z największą ilością zer i rzecz jasna może być dla każdego filtru różna. Selekcja zer, które będą zakwalifikowane jako nie-zera, w miarę możliwości dokonywana jest na zerach leżących blisko siebie, by zapewnić jak najbardziej ciągły dostęp do pamięci, co jest pożądane z punktu widzenia akceleracji w układach GPGPU. W celu reprezentacji wag w postaci macierzy rzadkiej, dla każdego filtru w każdej warstwie konwolucji budowana jest ich reprezentacja w formacie *Compressed sparse row (CSR)*, której przykład pokazuje rysunek 6.11. Do zapisu macierzy format ten potrzebuje następujących trzech tablic:

– *values* - zawiera ona nie zerowe elementy macierzy, których liczba oznaczana jest jako *nnz*,

– *coldix* - zawiera *nnz* liczb typu całkowitego, które są wskaźnikami na niezerowe wartości macierzy, tak że *coldix[i]* wskazuje kolumnę *i*-tego elementu w tablicy *values*. W przypadku obliczeń konwolucji wartości tej tablicy są modyfikowane tak, by zawierały od razu indeksy wejściowych danych, które będą użyte do obliczeń konwulucji (są to indeksy które odpowiadają nie zerowym wartościom filtrów). Funkcja *f(c, y, x)* dokonuje mapowania odpowiadającego indeksu (c, y, x) z macierzy, zawierającej dane wejściowe tak, że w przyjętym formacie CHW (channel, hihgh, weidth) $f(c, r, s) = (c * H + r)W + s$. Rozwiązanie to zostało zaproponowane w [75]. Dzięki niemu podczas obliczania konwolucji, zostaje zredukowany czas konieczny do obliczenia indeksów za każdym razem co ma wpływ na szybkość działania sieci,

– *rowptr* - zawiera elementy typu *integer*, któych liczba jest większa o jeden od liczby rzędów w oryginalnej macierzy, tak że *rowptr[i]* jest wskaźnikiem do pierwszego niezerowego elementu w *i*-tym kanale wyjściowym, a różnica *rowptr[i+1] - rowptr[i]* definiuje liczbę niezerowych elementów znajdujących się w tym wyjściowym kanale. Jak zostało to wcześniej wspomniane, w rozwiązaniu zaproponowanym w tej pracy różnica ta jest dla każdego kanału taka sama dzięki standaryzowaniu liczby zer dla każdego kanału wyjściowego, otrzymanych po operacji pruningu. Modyfikacja, jaką należy zrobić aby to osiągnąć, polega na oznaczeniu niektórych wartości zerowych jako niezerowych i tak je traktować podczas budowania formatu CSR. Zabieg ten nie powoduje zmiany wyniku oraz nie wymaga zagospodarowania dodatkowej ilości pamięci.

Aby zapisać macierz z *nnz* wartościami nie zerowymi o *M* rzędach wymagane jest zarezerwowanie $(2 \times nnz + M + 1) \times number\_of\_bytes\_for\_type$ bajtów. W związku z tym dla *rzadkości* macierzy wynoszącej $\sim 80\%$, używając formatu CSR, można zaoszczędzić $\sim 40\%$ pamięci, co jest niesłychanie ważne z punktu widzenia akceleratorów sprzętowych takich jak układy FPGA czy też używane w niniejszej pracy układy GPGPGU.

$$\begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 4 & 0 \\ 0 & 0 & 0 & 5 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 & 0 \\ 0 & 8 & 9 & 0 & 0 & 10 \end{bmatrix}$$

values = [1 2 3 4 5 6 7 8 9 10]

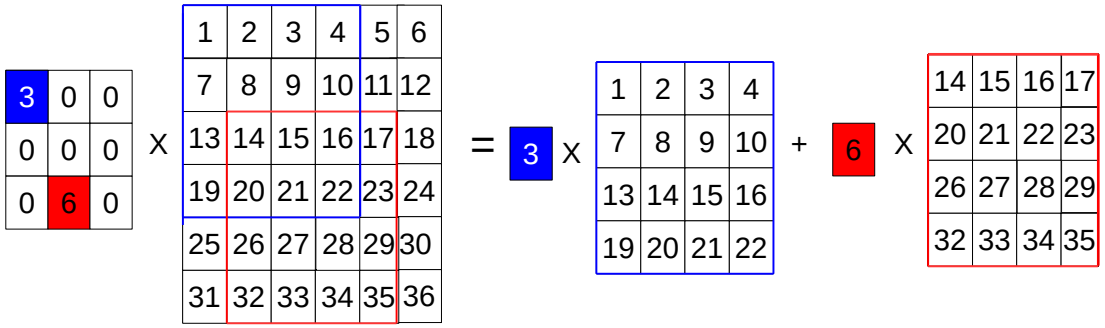
coldix = [0 1 2 4 3 5 3 1 2 5]

rowptr = [0 2 4 6 6 7 10]

Rysunek 6.11. Przykład reprezentacji macierzy w formacie CSR

6.4. Liczenie konwolucji przy użyciu operacji związanych z macierzami rzadkimi w układach GPGPU

Głównym celem obliczania konwolucji za pomocą operacji na macierzach rzadkich jest zredukowanie liczby mnożeń, gdyż możenie wartości przez zero nie ma wpływu na ostateczny wynik obliczeń, a co za tym idzie - może być pominięte. Z tego też względu najważniejszym punktem opisywanej metody jest zlokalizowanie indeksów, pod którymi w filtrach znajdują się wartości niezerowe, a następnie na tej podstawie obliczenie według wyżej wspomnianej formuły indeksów w macierzy z danymi wejściowymi, które będą poddane obliczeniom. Sytuacja ta została zobrazowana na rysunku 6.12, gdzie liczona jest konwolucja filtru o rozmiarach 3×3 (czyli takiego jaki jest używany w modelu VGG-16, rozdział 6.1.1), z wejściem o rozmiarach 6×6, gdzie liczba wejściowych kanałów wynosi dla ułatwienia jeden. Mając takie rozmiary danych, rezultatem będzie macierz 4×4. Filtr zawiera tylko dwie wartości nie zerowe, więc liczba wymaganych mnożeń w tym przypadku zostaje zredukowana z 9×16 do 2×16.



Rysunek 6.12. Obliczanie konwolucji przy użyciu operacji na macierzach rzadkich

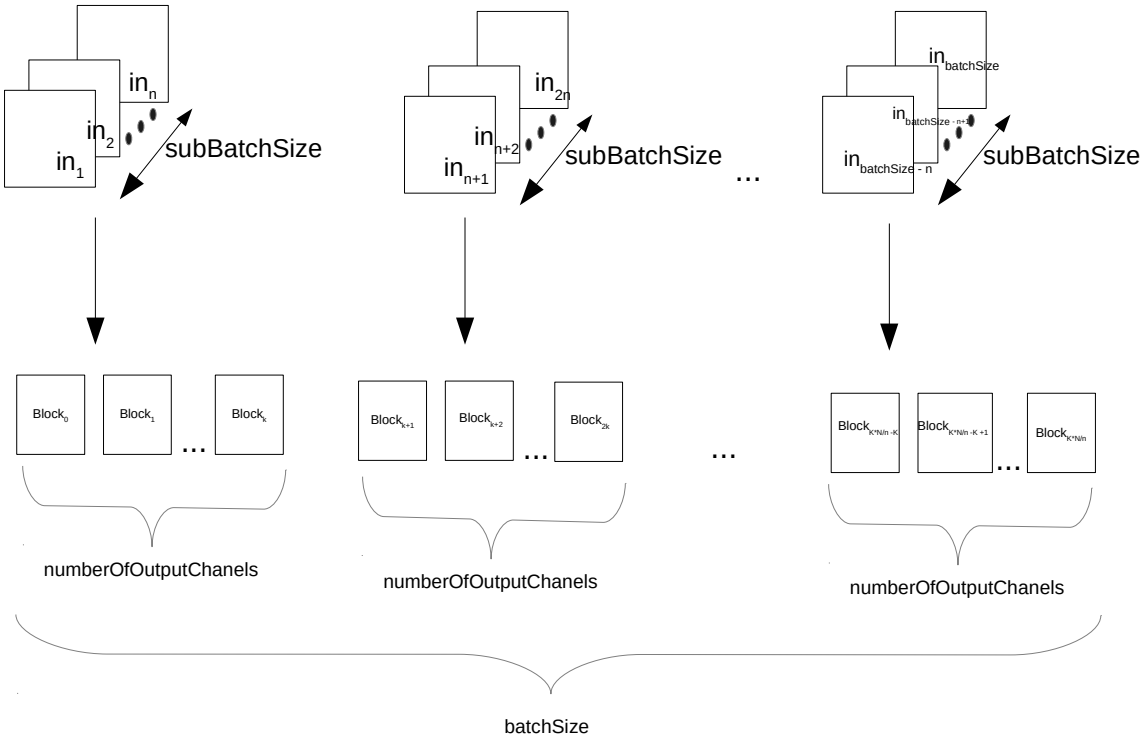
Mając wagi zapisane w formacie CSR, gdzie jak zostało wspomniane w tablicy *coldix* przechowywane są wcześniej obliczone przesunięcia jakich należy dokonać na macierzy z danymi wejściowymi, oraz dane zapisane w postaci NCHW, algorytm obliczający konwolucję, przy użyciu operacji związanych z macierzami rzadkimi przedstawia pseudo-kod z algorytmu 9.

Implementacja równoległa algorytmu w układach GPGPU, została zaczerpnięta z [10], gdzie każdy blok wątków odpowiedzialny jest za obliczenie pojedynczego kanału wyjściowego, więc dla jednego wektora wejściowego liczba bloków obliczających konwoulcję w danej warstwie będzie równa liczbie kanałów wyjściowych tej warstwy. W implementacji powstałej na potrzeby niniejszej pracy optymalizowana jest liczba próbek danych wejściowych, jakie będą przetwarzane przez tę liczbę bloków, a wartość ta będzie nazywana dalej jako $subBatchSize$. Wartość tego parametru dla każdej warstwy została ustalona w sposób eksperymentalny i jest zależna od jej wielkości. Jak się okazało należy ona do zbioru o $\{2, 4, 8\}$, co zostało dokładniej pokazane przy okazji prezentacji wyników. Optymalna wartość $subBatchSize$ nie jest jednakowa dla każdej warstwy, co jest spowodowane ograniczeniami związanymi z pamięcią podręczną (rozdział 3.2), która jest bardzo szybka, jednak w momencie jej zapełnienia dane zaczynają być przechowywane w pamięci globalnej karty (rozdział 3.2), do której dostęp jest mało efektywny. Dodatkowo należy sprawdzić czy dla danego algorytmu bardziej opłacalne jest zaangażowa-

nie większej ilości bloków czy też przetwarzanie przez mniejszą ilość większej liczby danych, przez co bloki mają więcej pracy ale jest ich mniej, co można ustalić jedynie eksperymentalnie (w dokumentacji każdej karty można znaleźć liczbę bloków, które będą przetwarzać się równolegle, ale doświadczenie pokazuje, że dla każdego algorytmu oraz dla każdego modelu karty graficznej, ustalenie optymalnej liczby bloków nie jest oczywiste i musi być sprawdzone eksperymentalnie). Prócz obliczeń teoretycznych, do wyznaczenia parametru $subBatchSize$ wykonano eksperymenty, dzięki czemu wyznaczono jego optymalna wartość dla każdej z warstw. Całkowita liczba bloków zaangażowanych w obliczanie konwolucji dla pojedynczej warstwy wynosi $\frac{batchSize*numberOfOutputChannel}{subBatchSize}$, co obrazuje rysunek 6.13.

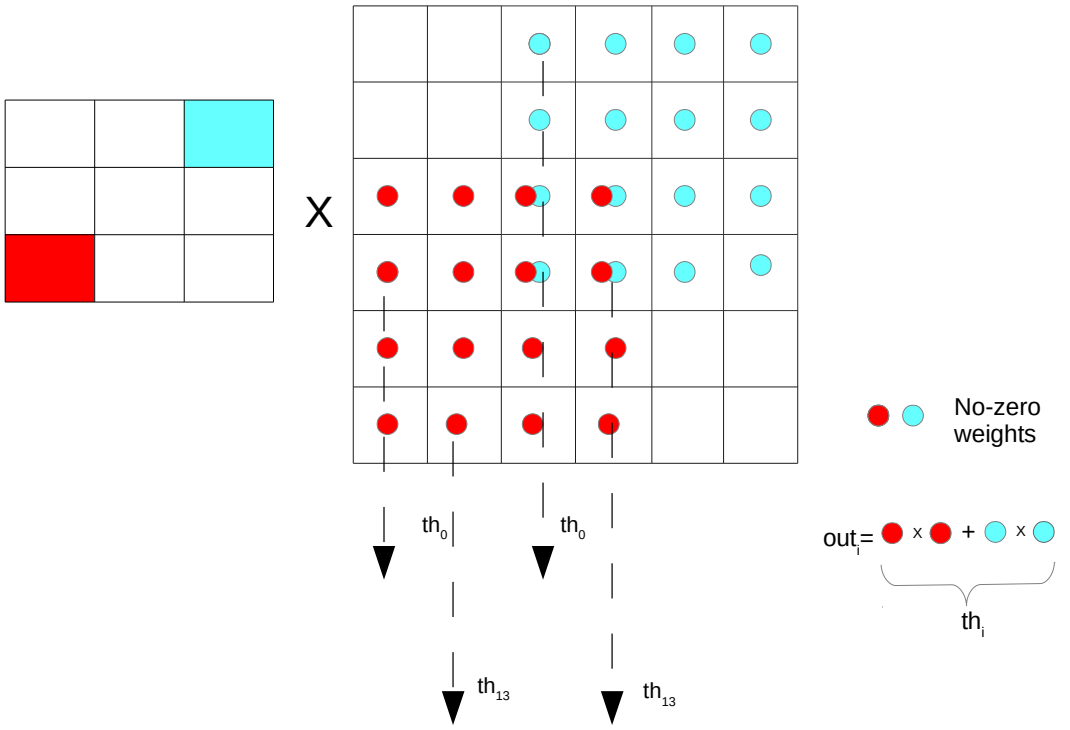
Algorytm 12. Sekwencyjna wersja metody konwolucji rzadkiej

```
1: function SPARCECONVOLUTION(input, coldix, rowptr, values, output)
2:     for n := 0 to N − 1 do
3:         for k := 0 to K − 1 do
4:             for j := rowptr[i] to rowptr[i + 1] do
5:                 value = values[j]
6:                 index = coldix[j]
7:                 for y := 0 to E − 1 do
8:                     for x := 0 to F − 1 do
9:                         out[n][k][y][x] += value*input[n][index]
   end function
```



Rysunek 6.13. Liczba bloków obliczających konwolucję dla danej warstwy

Dane wejściowe zapisane są w formacie 1D. Tablice pochodzące z formatu CSR, *value* oraz *coldix*, odpowiedzialne za wartości filtrów, przechowywane są w pamięci współdzielonej karty (ang. *shared memory*) (rozdział 3.2). Podczas obliczeń konkretne wartości filtru z tablicy *values* oraz indeksy z tablicy *coldix* wyciągane są z pamięci współdzielonej do pamięci wątku, a następnie są re-używane przy obliczeniach $subBatchSize$ wejściowych wektorów. Dzięki temu została ograniczona liczba odczytów z pamięci współdzielonej, co ma wpływ na akceleracje obliczeń. Podobnie do wartości niezerowych filtru oraz indeksów, cząstkowe wyniki przechowywane są również w rejestrach wątków, a po skończonych obliczeniach kopiowane są do pamięci globalnej, z której finalnie przenoszone są do procesora. Liczba wątków obliczających konwolucję w pojedynczym bloku, czyli przetwarzającą pojedynczy kanał wyjściowy, jest determinowana na podstawie rozmiaru konwolucji, a dokładniej mówiąc od wielkości jej wyjścia. Każdy pojedynczy wątek odpowiedzialny jest za obliczenie jednej wartości wyjściowej, czyli wykonuje mnożenie wartości wagi z odpowiadającą wartością z danych wejściowych, następnie otrzymany wynik sumuje z dotychczasowym wynikiem, który dla pierwszej wartości nie zerowej wynosi zero, a po skończonych obliczeniach kopiuje wynik do pamięci globalnej. Praca pojedynczego wątku jaką musi wykonać do obliczenia konwolucji została pokazana na rysunku 6.14, gdzie kolorami czerwonym i niebieskim zostały zaznaczone wartości nie zerowe filtru, a następnie przy ich użyciu zaznaczono w macierzy wejściowej odpowiadające im indeksy spod, których brane są wartości wejściowe. Przy takim podejściu każda wartość wyjściowa obliczana jest równolegle. Do ustalenia całkowitej liczby wątków w bloku brany jest pod uwagę poziom rzadkości, który w przypadku opisywanej implementacji jest taki sam dla wszystkich kanałów wyjściowych (rozdział 6.3), gdyż jest on zwykle większy niż wyjściowa wielkość splotu, stąd też więcej wątków jest potrzebnych by skopiować dane z pamięci globalnej do współdzielonej. Zarówno wektory z danymi wejściowymi jak również wagi zapisane w postaci CSR, oznaczone są jako *read-only* (w języku CUDA zapewniają to dyrektywy $cosnt\_\_restrict\_\_$ umieszczone przed typem danych), dzięki czemu możliwe jest użycie do ich przechowywania pamięci podręcznej (odpowiada za to funkcja ze środowiska CUDA $\_\_ldg$), co jest bardzo ważne w przypadku danych wejściowych, które nie mogą być skopiowane do pamięci współdzielonej, gdyż ta nie posiada wystarczająco miejsca. Dodatkowo zapewniony jest opisany w rozdziale 3.2 *global memory coalescing*. Funkcja obliczająca rozwiązanie została włączona we flow biblioteki cuDnn i jest uruchamiana w momencie, gdy został osiągnięty odpowiedni poziom rzadkości ($\sim$ 90% dla vgg-16 oraz 1x1 conv, $\sim$78% dla CNN-non static) dla warstw tych dla których przy takim poziomie zostało osiągnięte przyśpieszenie względem cuDnn. Prócz poziomu sparsity w niniejszej pracy sprawdzany jest dodatkowo wpływ użycia zredukowanej precyzji na szybkość obliczeń, zarówno przy użyciu cuDnn jak również zaproponowanej w tej pracy metody *konwolucji rzadkiej*. Aby tego dokonać dane wejściowe, jak również filtry, transformowane są z typu *float* do typu *half* przy użyciu dedykowanej biblioteki *Cuda-Math-Api*[98], a następnie przy użyciu funkcji z niej pochodzących, dokonywane są obliczenia matematyczne na 16-bitowych zmiennych.



Rysunek 6.14. Obliczanie konwolucji przy pomocy operacji związanych z macierzami rzadkimi

6.5. Dyskusja nad otrzymanymi wynikami

Wszystkie obliczenia zaprezentowane w tej sekcji, były uruchomione na karcie Nvidia Tesla V100-SXM2-32GB [96]. Zaprezentowane wyniki czasowe są średnią z dziesięciu uruchomień. Jak zostało wcześniej powiedziane cuDnn wybiera algorytm, którego użyje do przetworzenia konwolucji, więc prócz samych wyników czasowych, tabele zawierają informacje, który algorytm został wybrany przez cuDnn do obliczenia konwolucji. Pierwszą ciekawą rzeczą jest, zmiana algorytmu z *winograd* na *fft-tilling* wraz ze zwieszającym się rozmiarem *batcha*, co ma miejsce dla modelu VGG-16 dla warstw 3_1, 4_1, 4_2, 5_1, 5_2, 5_3. Wartość parametru $subBatchSize$ (rozdział 6.4), została ustalona w sposób eksperymentalny i tak dla trzech ostatnich warstw modelu VGG-16, obu warstw konwolucji 1×1 wynosi on 8, dla wszystkich pozostałych VGG-16 oraz dla warstwy z filterm 2 z CNN-non stati optymalne wyniki zostały uzyskane, gdy ten parametr był ustawiony na 4. Natomiast w przypadku CNN-non static dla warstwy z filtrem trzy parametr ten winien mieć wartość 2. Bez ustalenia wartości tego parametru metodą zaproponowaną w niniejszej pracy nie udałoby się uzyskać przyśpieszenia w stosunku do biblioteki cuDnn, gdyż w momencie ustalenia liczby bloków równej $numberOfOutputChannel * batchSize$ szybkość spadła dla VGG-16 oraz warstw $1x1$ o $\sim 10\%$, w przypadku CNN-non static był to spadek rzędu $\sim 12\%$. Jeszcze większy spadek zanotowano w momencie ustawienia liczby bloków na liczbę równą liczbie kanałów wyjściowych - wówczas dla wszystkich badanych warstw, odnotowano spadek między $\sim 38\%$ a $\sim 45\%$. Według dokumentacji liczba bloków jakie mogą równolegle działać na używanym modelu GPGPU wynosi 80, jednak jak się okazuje bardziej opłaca się uruchomić większą liczbę

Tabela 6.1. Wyniki czasowe VGG-16 warstwa 1_1(64x3x224x224)

Rozmiar *batch*	**Konwolucja rzadka float**	**Cudnn float \algorytm**	**Konwolucja rzadka half**	**Cudnn half \algorytm**
32	0.58	0.75\GEMM	0.55	0.72\GEMM
64	1.18	1.48\GEMM	1.12	1.4\GEMM
128	2.48	2.8\GEMM	2.21	2.71\GEMM

bloków niż przetwarzać w mniejszej ilości więcej danych. W przypadku programowania kart graficznych trzeba mieć na uwadze, że konkretny problem może mieć różne optymalne wartości liczby próbek przetwarzanych przez konkretną grupę bloków (w tym przypadku przez grupę rozumie się liczbę koniecznych bloków do przetworzenia konwolucji dla jednej próbki danych) dla różnych modeli GPGPU, dlatego też trzeba zawsze optymalizować dany algorytm pod konkretny model. Zaprezentowane wyniki, były zmierzone dla optymalnych wartości parametru $subBatchSize$. Dla modelu VGG-16 poziom rzadkości filtrów został ustawiony na $\sim 90\%$, gdyż był to najniższy poziom dla którego udało się uzyskać przyśpieszenie dla niektórych warstw z tego modelu. Jedynie dla pierwszej warstwy oraz dla trzech ostatnich, gdzie rozmiary konwolucji w formacie KCHW wynoszą odpowiednio $64 \times 3 \times 224 \times 224$ i $512 \times 512 \times 14 \times 14$, udało się uzyskać przyśpieszenie zarówno dla typu *float* oraz *half*. Dla wejściowej warstwy przyśpieszenie dla typu *float* wynosi $\sim 13\%$ oraz $\sim 12\%$ dla warstw ostatnich (oczywiście jest jednakowe dla trzech ostatnich warstw, gdyż mają one takie same rozmiary). Zmieniając typ danych z *float* na *half* uzyskane przyśpieszenia prezentują się następująco: $\sim 22\%$ - pierwsza warstwa oraz $\sim 11\%$ trzy ostatnie. Wartym podkreślenia jest fakt, że wspomniane warstwy są szybciej przetwarzane dla typu *half*, zarówno w przypadku *rzadkiej konwolucji* jak i dla biblioteki cuDnn. Fakt ten powinien być oczywistością, jednak dla biblioteki cuDnn nie jest, gdyż dla typu *half* biblioteka ta wybiera zawsze obliczanie splotu za pomocą metody *GEMMEM* (rozdział 6.2.2), co jest prawdopodobnie spowodowane chęcią wykorzystania opisanych w rozdziale 3.2.1 rdzeni typu *Tensor core*, dla których użycia konieczne jest posiadanie na wejściu danych typu *half* lub *int_8*. Jak się okazuje ten sposób liczenia konwolucji, przy użyciu cuDnn, na danych typu *half*, jest mniej efektywny dla warstw konwolucji z modelu VGG-16 z rozmiarem: $64 \times 64 \times 224 \times 224$, $256 \times 256 \times 56 \times 56$ and $512 \times 512 \times 28 \times 28$, od przetworzenia go na typie *float*, za pomocą algorytmu *FFT* (rozdział 6.2.3) lub *WINOGRAD* (rozdział 6.2.4). Biorąc pod uwagę jedynie wyniki osiągnięte dla typu *half*, podejście wykorzystujące macierze rzadkie może dodatkowo przyśpieszyć jeszcze warstwy $64 \times 64 \times 114 \times 114$, $256 \times 128 \times 58 \times 58$, $256 \times 256 \times 28 \times 28$, $512 \times 512 \times 28 \times 28$. Mając ten sam poziom rzadkości macierzy co dla VGG-16, możliwe jest uzyskanie lepszych wyników używając *konwolucji rzadkiej* zamiast cuDnn dla warstw konwolucyjnych 1×1 pochodzących z modelu ResNet (rozdział 6.1.2). Dla tego typu konwolucji cuDnn zawsze używa mnożenia macierzy, a wyniki dla parametru $N = 128$ (dla pozostałych proporcje były takie same) zostały zawarte w tabelach 6.12 oraz 6.13, gdzie uzyskane przyśpieszenia względem cuDnn, dla warstwy $256 \times 64 \times 1 \times 1$ dla typów *float* i *half* wynoszą odpowiednio $\sim 9\%$ oraz $\sim 11\%$, natomiast dla warstwy $64 \times 256 \times 1 \times 1$ jest to $\sim 19\%$ dla *float* oraz $\sim 20\%$ dla *half*.

Wysoka akceleracja została uzyskana konwolucji typu 1D, gdzie jako przykładu takiej konwolucji

Tabela 6.2. Wyniki czasowe VGG-16 warstwa 1_2(64x64x224x224)

Rozmiar *batch*	**Konwolucja rzadka float**	**Cudnn float \algorytm**	**Konwolucja rzadka half**	**Cudnn half \algorytm**
32	14.098	5.54\WINOGRAD	7.44	9.48\GEMM
64	28.47	11.41\WINOGRAD	14.23	14.09\GEMM
128	60.73	19.07\WINOGRAD	27.08	31.82\GEMM

Tabela 6.3. Wyniki czasowe VGG-16 warstwa 2_1(128x64x112x112)

Rozmiar *batch*	**Konwolucja rzadka float**	**Cudnn float \algorytm**	**Konwolucja rzadka half**	**Cudnn half \algorytm**
32	4.29	2.66\WINOGRAD	2.14	2.42\GEMM
64	8.58	4.67\WINOGRAD	4.12	4.29\GEMM
128	16.45	10.56\WINOGRAD	8.87	9.48\GEMM

użyto modelu *CNN-non static* (rozdział 6.1.3), który przeznaczony jest do klasyfikacji tekstu. W tym przypadku najbardziej znaczące przyśpieszenie względem cuDnn, który w dla tej konwolucji podobnie jak dla konwolucji 1×1 zawsze wybiera mnożenie macierzy, zostało osiągnięte dla warstwy konwolucyjnej z wielkością kernela 2, gdzie wystarczający poziom rzadkości wynosi $\sim 77\%$ by uzyskać $\sim 9\%$ oraz $\sim 11\%$ przyspieszenia odpowiednio dla typów *float* i *half*. Warty podkreślenia jest fakt, że wspominany poziom rzadkości jest często osiągalny dla danych tekstowych, przy okazji operacji *Pruningu*, co znaczy że nie odnotowuje się znaczącego spadku efektywności. Co więcej, pozostając na wysokim poziomie skuteczności sieci, możliwe jest uzyskanie tego współczynnika na poziomie $\sim 90\%$, gdzie w takim przypadku przyśpieszenie liczenia konwolucji za pomocą operacji na macierzach rzadkich względem cuDnn wynosi ok. 2 razy. Świadczy to o tym, że zaproponowana w niniejszej pracy metoda liczenia konwolucji jest warta rozważenia przy przetwarzaniu konwolucji typu 1D, gdyż osiąga one duże przyśpieszenie względem cuDnn, tym bardziej, że w praktyce możliwe jest uzyskanie wymaganego poziomu rzadkości. Wyniki czasowe przedstawiające opisaną sytuację zostały zawarte w tabelach: 6.10 i 6.11, gdzie liczba danych wejściowych była ustawiona na 128, gdyż dla pozostałych (32, 64, 256) proporcje były zachowane. Ostatnią wartą podkreślania rzeczą, jest wpływ zaproponowanego w tej pracy sposobu budowania macierzy CSR, czyli ustalenia w sztuczny sposób (rozdział 6.3) jednakowego poziomu rzadkości dla wszystkich wyjściowych kanałów danej warstwy. I tak w przypadku VGG-16 oraz warstw typu 1×1 z modelu ResNet spowodowało to przyśpieszenie rzędu $\sim 28\%$, natomiast w przypadku CNN-non static akceleracja wyniosła $\sim 26\%$. Jest to głównie spowodowane faktem, że wartość ta znana jest na etapie kompilacji, dzięki czemu kompilator jest wstanie zoptymalizować niektóre operacje oraz nie musi jej obliczać dla każdego filtru wyjściowego z osobna.

Tabela 6.4. Wyniki czasowe VGG-16 warstwa 2_2(128x128x112x112)

Rozmiar *batch*	**Konwolucja rzadka float**	**Cudnn float \algorytm**	**Konwolucja rzadka half**	**Cudnn half \algorytm**
32	7.94	4.34\WINOGRAD	4.78	4.68\GEMM
64	14.02	8.66\WINOGRAD	8.9	8.37\GEMM
128	28.11	17.28\WINOGRAD	17.31	15.88\GEMM

Tabela 6.5. Wyniki czasowe VGG-16 warstwa 3_1 (256x128x56x56)

Rozmiar *batch*	**Konwolucja rzadka float**	**Cudnn float \algorytm**	**Konwolucja rzadka half**	**Cudnn half \algorytm**
32	3.72	2.78\WINOGRAD	2.3	2.35\GEMM
64	7,14	4.65\FFT-TILING	4.53	4.63\GEMM
128	13,6	9.21\FFT-TILING	8.74	7.81\GEMM

Tabela 6.6. Wyniki czasowe VGG-16 warstwa 3_2 (256x256x56x56)

Rozmiar *batch*	**Konwolucja rzadka float**	**Cudnn float \algorytm**	**Konwolucja rzadka half**	**Cudnn half \algorytm**
32	6.96	4.01\FFT_TILING	4.23	4.63\GEMM
64	13.7	7.27\FFT-TILING	7.27	8.39\GEMM
128	23.72	14.27\FFT_TILING	15.8	16.09\GEMM

Tabela 6.7. Wyniki czasowe VGG-16 warstwa 4_1 (512x256x28x28)

Rozmiar *batch*	**Konwolucja rzadka float**	**Cudnn float \algorytm**	**Konwolucja rzadka half**	**Cudnn half \algorytm**
32	3.2	2.18\Winograd	1.66	2.32\GEMM
64	5.72	3.66\FFT-TILING	3.2	4.2\GEMM
128	9.34	6.7\FFT-TILING	6.1	7.8\GEMM

Tabela 6.8. Wyniki czasowe VGG-16 warstwa 4_2 & 4_3 (512x512x28x28)

Rozmiar *batch*	**Konwolucja rzadka float**	**Cudnn float \algorytm**	**Konwolucja rzadka half**	**Cudnn half \algorytm**
32	4.46	5.4\Winograd	3.94	4.62\GEMM
64	8.69	8.3\FFT-TILING	6.98	9.2\GEMM
128	16.06	15.02\FFT-TILLING	14.01	16.82\GEMM

Tabela 6.9. Wyniki czasowe VGG-16 warstwa 5_1 & 5_2 & 5_3 (512x512x14x14)

Rozmiar *batch*	**Konwolucja rzadka float**	**Cudnn float \algorytm**	**Konwolucja rzadka half**	**Cudnn half \algorytm**
32	1.11	1.54\Winograd	1.06	1.43\GEMM
64	2.18	2.96\FFT	2.12	2.78\GEMM
128	4.31	4.8\FFT	4.18	4.66\GEMM

Tabela 6.10. CNN-non-static dla wejścia 300x64 z rozmiarem kernela 2

Typ danych	**CUDNN**	**Konwolucja rzadka**		
		77%	**83%**	**87,5%**
float	0.1924	0.176	0.126	0.102
half	0.1602	0.145	0.097	0.069

Tabela 6.11. CNN-non-static dla wejścia 300x64, z rozmiarem kernela 3

Typ danych	**CUDNN**	**Konwolucja rzadka**		
		77%	**83%**	**87,5%**
float	0.231	0.236	0.188	0.135
half	0.204	0.182	0.148	0.103

Tabela 6.12. Wyniki czasowe dla ResNet 256x64x1x1

Typ danych	**CUDNN**	**Konwolucja rzadka**
float	0.35	0.32
half	0.30	0.27

Tabela 6.13. Wyniki czasowe dla ResNet 64x256x1x1

Typ danych	**CUDNN**	**Konwolucja rzadka**
float	0.37	0.31
half	0.29	0.24

7. Podsumowanie

Celem prac badawczych przedstawionych w niniejszej rozprawie była akceleracja wybranych algorytmów uczenia maszynowego oraz wybranych populacyjnych algorytmów inteligencji obliczeniowej poprzez ich implementację w układach GPGPU. Dodatkowo w algorytmach uczenia maszynowego, zbadano wpływ użycia zredukowanej precyzji danych na szybkość obliczeń oraz oszczędność pamięci. W konsekwencji opracowano szereg nowych implementacji, które okazały się bardziej efektywne od dotychczas istniejących rozwiązań.

Pierwszą grupą algorytmów dla której zostały podjęte próby akceleracyjne były algorytmy z rodziny algorytmów inteligencji obliczeniowej. Pierwszym algorytmem zaimplementowanym w procesorach graficznych, był algorytm *SDLS*, rozwiązujący problem *LABS*, co stanowi innowacyjne podejście, gdyż w dotychczasowych badaniach można było spotkać jego implementacje na procesorach ogólnego przeznaczenia [29] oraz w układach FPGA [91]. Algorytm *SDLS* należy do rodziny problemów NP-trudnych, będący algorytmem silnie równoległym, stąd też istniała potrzeba użycia do jego rozwiązania akceleratorów sprzętowych dedykowanych do obliczeń równoległych. Dzięki użyciu kart graficznych dla zaproponowanych wielkości problemu, uzyskano kilku-dziesięciokrotne przyśpieszenia względem równoległej implementacji na procesorach ogólnego przeznaczenia (zaprezentowane wyniki dotyczą długości ciągów dużo większych od tych zaprezentowanych dla układów FPGA w [91], stąd też nie ma możliwości dokonania porównania obu implementacji). Kolejnym algorytmem rozwiązującym problem *LABS*, który został zaimplementowany w układach GPGPU jako nowe podejście na potrzeby tej pracy, jest algorytm *TABU*. W tym przypadku przyśpieszenia względem procesora utrzymały się na takim samym poziomie jak w przypadku algorytmu *SDLS*. Dodatkowo moduły sprzętowe obliczające *SDLS* oraz *Tabu*, stały się częścią hybrydowego algorytmu *EMAS* [83]. Podejście takie zaowocowało szybszym uzyskaniem optymalnego rezultatu w przypadku *SDLS* niż w wersji, gdzie całość algorytmu była wykonywana na procesorze. Natomiast koncepcja EMAS, używająca jako lokalnej optymalizacji algorytmu *Tabu* w wersji sprzętowej, pozwoliła uzyskać optima, których nie udało się uzyskać w wersji nie używającej układu GPGPU. Tak imponujące osiągnięcia procesora graficznego dla przedstawionego problemu, pozwoliły na zaproponowanie dwóch nowych algorytmów, wcześniej nie spotykanych w literaturze nazwanych *SDLS-2* oraz *SDLS z przeszukiwaniem w głąb*. Rozwiązania te, jak wskazuje sama nazwa, są rozszerzeniem standardowej metody *SDLS*. Ze względu na ich wysoką złożoność obliczeniową oraz na wysoką efektywność sprzętowej wersji algorytmu *SDLS*, zostały one zaimplementowane jedynie na kartach graficznych. Skuteczność nowych rozwiązań została zbadana poprzez

porównanie ich ze standardową metodą *SDLS*. W tym celu każdy z algorytmów został uruchomiony na okres jednej godziny, gdzie po każdorazowym ukończeniu poszukiwań przez dany kernel, następowało losowanie nowych danych wejściowych i poszukiwania rozpoczynały się od nowa. Eksperymenty pokazały wysoką skuteczność algorytmu *SDLS z przeszukiwaniem w głąb*, który był w stanie znaleźć najlepsze rozwiązania dla wszystkich zaproponowanych długości ciągów. Fakt ten napawa optymizmem przed włączeniem tego algorytmu do koncepcji *EMAS*, gdzie wspomniany pierwiastek losowy jest zastępowany przez bardziej inteligentne metody [83], co pozwala przypuszczać, że uda się osiągnąć jeszcze bardziej optymalne rezultaty. Należy wspomnieć, że owe modyfikacje można wprowadzić do algorytmu *Tabu* (dzięki któremu możliwe było znalezienie bardziej optymalnych rozwiązań niż w przypadku *SDLS*), co pozwala przypuszczać, że jego ulepszone wersje będą jeszcze bardziej efektywne. Dużą rolę w rozwiązywaniu problemu *LABS* przy użyciu opisanych technik, odegrała zredukowana prezencja, gdyż dzięki użyciu typu *int_8* (wprowadzanemu w wersji CUDA 7.5), do przechowywania ciągów wejściowych, możliwe stało się rozwiązanie problemu dla długości ciągów dla których przy użyciu podstawowego typu *int* było by nie możliwe ze względu na ograniczenia pamięciowe.

Kolejną grupą algorytmów, które zostały poddane akceleracji były algorytmy uczenia maszynowego. Algorytmem na podstawie, którego udowodniano stwierdzenie, że „przeprowadzenie obliczeń przy użyciu zredukowanej precyzji nie zawsze prowadzi do spadku wydajności algorytmu" był algorytm wektorów nośnych (SVM). Zadaniem, do którego został użyty algorytm SVM, była klasyfikacja tekstu. W pracy zaprezentowane zostały dwie propozycje metod dokonujących kwantyzacji danych w procesie trenowania SVM, dzięki którym dla dwóch zbiorów uczących możliwe było przeprowadzenie uczenia przy użyciu 6 bitów, nie odnotowując znaczącego spadku jego skuteczności ($\sim 1\%$.). Implementacja sprawdzająca wpływ szerokości danych na efektywność algorytmu została wykonana w języku C++ i miała za zadanie dostarczyć informację jak „nisko" można zejść z szerokością danych by algorytm działał poprawnie. Mając wiedzę, że przy użyciu 16 bitów (16 bitów zostało użyte do sprawdzenia efektywności algorytmów, gdyż jest to najmniejszy typ nie całkowitoliczbowy wspierany przez środowisko CUDA) algorytm dokonuje dostatecznie dobrej klasyfikacji, sprawdzono jaki wpływ na czas uczenia będzie miało użycie takiej szerokości w procesie trenowania zaimplementowanym na procesorach graficznych. Na początku dokonano pomiaru przyśpieszenia jaki może zostać uzyskany dla podstawowych operacji macierzowych, które są podstawą w procesie trenowania wektorów nośnych, a które zarazem stanowią ważną część w innych algorytmach uczenia maszynowego (np. w sieciach neuronowych). Jak się okazało użycie typu *half* (16-bitów) powoduje szybsze o kilkanaście procent wykonanie obliczeń dla operacji mnożenia dwóch wektorów oraz obliczania produktu skalarnego dwóch wektorów w stosunku do typu *float*. Kilkadziesiąt procent przyśpieszenia udało się osiągnąć dla najbardziej kosztownej operacji wykorzystywanej w procesie treningu czyli mnożeniu macierzy przez wektor. Cały proces trenowania na układach GPGPU, przy użyciu zredukowanej precyzji, trwał o $\sim 24\%$ krócej od tego przeprowadzanego przy użyciu pojedynczej precyzji.

Kolejnym ważnym aspektem podjętym na potrzeby niniejszej pracy była próba implementacji bardziej efektywnej wersji obliczania konwolucji od tej oferowanej przez bibliotekę *cuDnn*, uważaną

za najbardziej optymalne narzędzie służące do przetwarzania sieci neuronowych na procesorach graficznych. Nowe podejście opiera się na wykorzystaniu operacji związanych z przetwarzaniem macierzy rzadkich czyli takich, gdzie w elementach jednej macierzy zdecydowana większość elementów ma wartość równą zero. Taka sytuacja często zachodzi w sieciach splotowych dla macierzy przechowujących wartości wag poszczególnych warstw, gdyż te poddawane są operacji *Pruningu*. W pracy zaproponowano nowe podejście transformujące macierze wag do postaci macierzy rzadkich w formie *CSR*. Sposób ten wykorzystuje informację o ilości wag z wartością zero, będącymi efektem *Pruningu*. Informację tę wykorzystano do ujednolicenia poziomu rzadkości dla wszystkich kanałów wyjściowych danej warstwy. Dzięki temu możliwe stało się uniknięcie obliczania poziomu rzadkości podczas samych kalkulacji, gdyż ten był jednakowy oraz znany przed ich rozpoczęciem. Dodatkowo zbadano wpływ użycia zredukowanej precyzji danych (w tym przypadku podobnie jak w SVM był to typ *half*, przechowujący dane na 16 bitach) zarówno dla zaproponowanej metody nazwanej *konwolucją rzadką* jak również dla tych pochodzących z biblioteki *cuDnn*. W wyniku powyższego udowodniono, że używając mniejszej ilości bitów do reprezentowania danych, w przypadku metody *konwolucji rzadkiej*, następuje spora poprawa rezultatów czasowych, co nie zawsze można zaobserwować w przypadku biblioteki *cuDnn*. Eksperymenty te mają wysoką wartość, gdyż obecnie prowadzonych jest wiele badań [38][39][125] mających na celu dokonanie kwantyzacji danych podczas treningu sieci konwolucyjnych z jednoczesnym brakiem spadku ich skuteczności. Cały algorytm wraz z zaistnieniem określonych warunków (wielkość warstwy, poziom rzadkości), okazuje się zdolny do szybszego obliczenia operacji konwolucji od algorytmów udostępnionych przez bibliotekę *cuDnn*. W związku z tym w momencie wystąpienia odpowiednich okoliczności może on być wykorzystany do procesowania konowloucji w miejsce dedykowanej biblioteki.

Do najważniejszych oryginalnych rozwiązań autora zaprezentowanych w tej pracy, należy zaliczyć:

- wykonanie w pełni sprzętowej implementacji algorytmu *SDLS* rozwiązującego problem *LABS*,
- wykonanie w pełni sprzętowej implementacji algorytmu *Tabu* rozwiązującego problem *LABS*,
- opracowanie oraz implementacja sprzętowa, nowatorskiej metody rozwiązującej problem *LABS*, nazwanej *SDLS-2*,
- opracowanie oraz implementacja sprzętowa, nowatorskiej metody rozwiązującej problem *LABS*, nazwanej *SDLS z przeszukiwaniem w głąb*,
- zbadanie wpływu użycia metody dokonującej kwantyzacji danych do danej szerokości, nazwanej *max magnitude dynamic fixed-point quantization*, podczas treningu algorytmu wektorów nośnych na jego skuteczność,
- zbadanie wpływu użycia metody dokonującej kwantyzacji danych do danej szerokości, nazwanej *min-max dynamic fixed-point quantization*, podczas treningu algorytmu wektorów nośnych na jego skuteczność,

– zbadanie wpływu użycia zredukowanej precyzji danych na szybkość procesu trenowania algorytmu wektorów nośnych, w implementacji sprzętowej na układach GPGPU,

– opracowanie oraz implementacja efektywnej metody obliczania konwolucji, wykorzystującej operacje na macierzach rzadkich,

– zbadanie wpływu użycia zredukowanej precyzji danych na szybkość wykonywania operacji konwolucji zarówno przez zaproponowaną metodę jak również metody pochodzące z dedykowanej biblioteki na procesorach graficznych,

Wyniki przedstawione w pracy upoważniają do stwierdzenia, że postawiona teza:
„Implementacja algorytmów inteligencji obliczeniowej oraz uczenia maszynowego w akceleratorach GPGPU prowadzi do przyśpieszenia ich wykonania w stosunku do implementacji CPU. Użycie w implementacji sprzętowej zredukowanej precyzji danych, poprawia szybkość kart graficznych, nie zawsze prowadząc do pogorszenia jakości algorytmu”
została w pełni udowodniona.

Implementacje wybranych algorytmów zaprezentowanych w pracy pokazują, że akceleratory graficzne znakomicie nadają się do efektywnej implementacji popularnych i szeroko wykorzystywanych algorytmów z dziedziny sztucznej inteligencji. Należy zwrócić uwagę, iż ta grupa algorytmów posiada wewnętrzną równoległość, z tego też względu karty graficzne wydają się naturalnym wyborem w celach akceleracyjnych. Przedstawione rezultaty pokazują jak potężnym narzędziem akceleracyjnym są procesory graficznie, bez których w wielu przypadkach nie możliwe stało by się przeprowadzenie wartościowych obliczeń, ze względu na ich wysoki poziom złożoności obliczeniowej. Prowadzone prace można uznać szczególnie przydatne we wszystkich aplikacjach, w których aktualne osiągnięcia istniejących rozwiązań są zbyt mało efektywne, by sprostać postawionymi przed nimi wymaganiami czasowymi. Dynamiczny rozwój układów GPGPU pozwala postawić twierdzenie, że w niedalekiej przyszłości układy te będą nieodzowną częścią każdego systemu wymagającego dostępu do dużej mocy obliczeniowej, a przedstawione w tej pracy rozwiązania mogą posłużyć jako rozszerzenie istniejących rozwiązań (cuDnn) bądź zalążek nowej biblioteki umożliwiającej przeprowadzenie obliczeń z dziedziny inteligencji obliczeniowej.

Bibliografia

[1] Karel Adámek i in. *GPU Fast Convolution via the Overlap-and-Save Method in Shared Memory*. Paź. 2019.

[2] Michael Bartholomew–Biggs. "The Steepest Descent Method". *Nonlinear Optimization with Engineering Applications*. Boston, MA: Springer US, 2008, s. 1–8. ISBN: 978-0-387-78723-7. DOI: 10.1007/978-0-387-78723-7_7. URL: http://dx.doi.org/10.1007/978-0-387-78723-7_7.

[3] Christian Bauckhage i Daniel Speicher. *Lecture Notes on Machine Learning: The Karush-Kuhn-Tucker Conditions (Part 1)*. Sierp. 2019.

[4] Piotr Bojanowski i in. "Enriching Word Vectors with Subword Information". *Transactions of the Association for Computational Linguistics* 5 (2017), s. 135–146. DOI: 10.1162/tacl_a_00051. URL: https://www.aclweb.org/anthology/Q17-1010.

[5] Borko Boskovic, Franc Brglez i Janez Brest. "Low-Autocorrelation Binary Sequences: on the Performance of Memetic-Tabu and Self-Avoiding Walk Solvers". *CoRR* abs/1406.5301 (2014). arXiv: 1406.5301. URL: http://arxiv.org/abs/1406.5301.

[6] Borko Bošković, Franc Brglez i Janez Brest. "Low-autocorrelation binary sequences: On improved merit factors and runtime predictions to achieve them". *Applied Soft Computing* 56 (2017), s. 262–285. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2017.02.024. URL: http://www.sciencedirect.com/science/article/pii/S1568494617301023.

[7] J. Brest i B. Boskovic. "In Searching of Long Skew-symmetric Binary Sequences with High Merit Factors". *ArXiv* abs/2011.00068 (2020).

[8] Z. Cao, L. Liu i O. Markowitch. "Comment on "Highly Efficient Linear Regression Outsourcing to a Cloud"". *IEEE Transactions on Cloud Computing* 7.3 (2019), s. 893–893.

[9] Krzysztof Cetnarowicz, Marek Kisiel-Dorohinicki i Edward Nawarecki. "The Application of Evolution Process in Multi-Agent World to the Prediction System" (sty. 1996).

[10] Xuhao Chen. "Escoin: Efficient Sparse Convolutional Neural Network Inference on GPUs" (2018). arXiv: 1802.10280 [cs.DC].

[11] Sharan Chetlur i in. *cuDNN: Efficient Primitives for Deep Learning*. 2014. arXiv: 1410.0759 [cs.NE].

[12] Kyunghyun Cho i in. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[13] Corinna Cortes i Vladimir Vapnik. "Support-Vector Networks". *Mach. Learn.* 20.3 (wrz. 1995), s. 273–297. ISSN: 0885-6125. DOI: 10.1023/A:1022627411411. URL: https://doi.org/10.1023/A:1022627411411.

[14] Jacob Devlin i in. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". *ArXiv* abs/1810.04805 (2019).

[15] *Dokumentacja architektury KEPLER*. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf.

[16] *Dokumentacja Cooperative Groups API*. https://developer.nvidia.com/blog/cooperative-groups/.

[17] Zurek Dominik i in. "Comparison Of Hybrid Sorting Algorithms Implemented On Different Parallel Hardware Platforms". *Computer Science* 14 (sty. 2013), s. 679. DOI: 10.7494/csci.2013.14.4.679.

[18] Jack Dongarra i in. "The Design and Performance of Batched BLAS on Modern High-Performance Computing Systems". *Procedia Computer Science* 108 (2017). International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland, s. 495–504. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2017.05.138. URL: http://www.sciencedirect.com/science/article/pii/S1877050917307056.

[19] Iván Dotú i Pascal Van Hentenryck. "A Note on Low Autocorrelation Binary Sequences". *Principles and Practice of Constraint Programming - CP 2006: 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006. Proceedings*. Red. Frédéric Benhamou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, s. 685–689. ISBN: 978-3-540-46268-2.

[20] John Duchi, Elad Hazan i Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". *J. Mach. Learn. Res.* 12.null (lip. 2011), s. 2121–2159. ISSN: 1532-4435.

[21] Mohamed Elleuch, Rania Maalej i Monji Kherallah. "A New Design Based-SVM of the CNN Classifier Architecture with Dropout for Offline Arabic Handwritten Recognition". *Procedia Computer Science* 80 (2016), s. 1712–1723. DOI: 10.1016/j.procs.2016.05.512. URL: https://doi.org/10.1016%2Fj.procs.2016.05.512.

[22] R. M. Esteves, T. Hacker i C. Rong. "Competitive K-Means, a New Accurate and Distributed K-Means Algorithm for Large Datasets". *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. T. 1. 2013, s. 17–24.

[23] Gary Flake i Steve Lawrence. "Efficient SVM regression training with SMO". *Machine Learning* 46 (mar. 2001). DOI: 10.1023/A:1012474916001.

[24] M. J. Flynn. "Very high-speed computing systems". *Proceedings of the IEEE* 54.12 (1966), s. 1901–1909.

[25] David Fogel. "Artificial Intelligence through Simulated Evolution". *Evolutionary Computation: The Fossil Record* (list. 2009), s. 227–296. DOI: 10.1109/9780470544600.ch7.

[26] Stan Franklin i Art Graesser. "Is It an agent, or just a program?: A taxonomy for autonomous agents". *Intelligent Agents III Agent Theories, Architectures, and Languages*. Red. Jörg P. Müller, Michael J. Wooldridge i Nicholas R. Jennings. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, s. 21–35. ISBN: 978-3-540-68057-4.

[27] Fuming Lin i J. Guo. "A novel support vector machine algorithm for solving nonlinear regression problems based on symmetrical points". *2010 2nd International Conference on Computer Engineering and Technology*. T. 2. 2010, s. V2-176-V2–180.

[28] Jose E. Gallardo, Carlos Cotta i Antonio J. Fernandez. "A Memetic Algorithm for the Low Autocorrelation Binary Sequence Problem". *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*. GECCO '07. London, England: ACM, 2007, s. 1226–1233. ISBN: 978-1-59593-697-4.

[29] José E. Gallardo, Carlos Cotta i Antonio J. Fernández. "Finding Low Autocorrelation Binary Sequences with Memetic Algorithms". *Appl. Soft Comput.* 9.4 (wrz. 2009), s. 1252–1262. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2009.03.005. URL: http://dx.doi.org/10.1016/j.asoc.2009.03.005.

[30] Michel Gendreau i Jean-Yves Potvin. *Handbook of Metaheuristics*. 2nd. Springer Publishing Company, Incorporated, 2010. ISBN: 1441916636, 9781441916631.

[31] Xavier Glorot i Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*. 2010.

[32] Fred Glover i Manuel Laguna. *Tabu Search*. Norwell, MA, USA: Kluwer Academic Publishers, 1997. ISBN: 079239965X.

[33] M. Golay. "Sieves for low autocorrelation binary sequences". *IEEE Transactions on Information Theory* 23.1 (sty. 1977), s. 43–51. ISSN: 0018-9448.

[34] M. Golay. "The merit factor of long low autocorrelation binary sequences (Corresp.)" *IEEE Transactions on Information Theory* 28.3 (maj 1982), s. 543–549. ISSN: 0018-9448.

[35] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0201157675.

[36] Ian Goodfellow, Yoshua Bengio i Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[37] Gongde Guo i in. "KNN Model-Based Approach in Classification" (sierp. 2004).

[38] Mo'taz Al-Hami i in. "Methodologies of Compressing a Stable Performance Convolutional Neural Networks in Image Classification". *Neural Processing Letters* 51 (2019), s. 105–127.

[39] Motaz Al-Hami i in. "Towards a Stable Quantized Convolutional Neural Networks: An Embedded Perspective". Sty. 2018, s. 573–580. DOI: 10.5220/0006651305730580.

[40] Song Han i in. "Learning both Weights and Connections for Efficient Neural Networks". *CoRR* abs/1506.02626 (2015). arXiv: 1506.02626. URL: http://arxiv.org/abs/1506.02626.

[41] K. He i in. "Deep Residual Learning for Image Recognition". *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, s. 770–778.

[42] K. He i in. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, s. 1026–1034.

[43] Yang He i in. "Soft Filter Pruning for Accelerating Deep Convolutional Neural Networks". *CoRR* abs/1808.06866 (2018). arXiv: 1808.06866. URL: http://arxiv.org/abs/1808.06866.

[44] Yihui He, Xiangyu Zhang i Jian Sun. "Channel Pruning for Accelerating Very Deep Neural Networks". *CoRR* abs/1707.06168 (2017). arXiv: 1707.06168. URL: http://arxiv.org/abs/1707.06168.

[45] John L. Hennessy i David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X.

[46] John H. Holland. "Outline for a Logical Theory of Adaptive Systems". *J. ACM* 9.3 (lip. 1962), s. 297–314. ISSN: 0004-5411. DOI: 10.1145/321127.321128. URL: https://doi.org/10.1145/321127.321128.

[47] Thorsten Joachims. "Text Categorization with Support Vector Machines: Learning with Many Relevant Features". *Proceedings of the 10th European Conference on Machine Learning*. ECML'98. Chemnitz, Germany: Springer-Verlag, 1998, s. 137–142. ISBN: 3540644172. DOI: 10.1007/BFb0026683. URL: https://doi.org/10.1007/BFb0026683.

[48] Jun Tang. "A color image segmentation algorithm based on region growing". *2010 2nd International Conference on Computer Engineering and Technology*. T. 6. 2010, s. V6-634-V6–637.

[49] V. Kecman i T. Yang. "Adaptive Local Hyperplane for regression tasks". *2009 International Joint Conference on Neural Networks*. 2009, s. 1566–1570.

[50] Yoon Kim. "Convolutional Neural Networks for Sentence Classification". *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, paź. 2014, s. 1746–1751. DOI: 10.3115/v1/D14-1181. URL: https://www.aclweb.org/anthology/D14-1181.

[51] Diederik P. Kingma i Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[52] M. Kisiel-Dorohinicki i Akademia Górniczo-Hutnicza im. Stanisława Staszica (Kraków). *Agentowe architektury populacyjnych systemów inteligencji obliczeniowej*. Rozprawy, Monografie - Akademia Górniczo-Hutnicza im. Stanisława Staszica. Wydawnictwa Akademii Górniczo-Hutniczej im. Stanisława Staszica, 2013. ISBN: 9788374645881. URL: https://books.google.pl/books?id=oEMlnwEACAAJ.

[53] Marek Kisiel-Dorohinicki. "Agent-Oriented Model of Simulated Evolution". *SOFSEM 2002: Theory and Practice of Informatics*. Red. William I. Grosky i František Plášil. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, s. 253–261. ISBN: 978-3-540-36137-4.

[54] Marek Kisiel-Dorohinicki. "Flock-Based Architecture for Distributed Evolutionary Algorithms". Czer. 2004, s. 841–846. DOI: 10.1007/978-3-540-24844-6_130.

[55] Magdalena Kolybacz i in. "Efficiency Of Memetic And Evolutionary Computing In Combinatorial Optimisation." *ECMS*. Red. Webjørn Rekdalsbakken, Robin T. Bye i Houxiang Zhang. European Council for Modeling i Simulation, 2013, s. 525–531. ISBN: 978-0-9564944-6-7.

[56] Michał Kowol, Aleksander Byrski i Marek Kisiel-Dorohinicki. "Agent-based Evolutionary Computing for Difficult Discrete Problems". *Procedia Computer Science* 29 (2014), s. 1039–1047. ISSN: 1877-0509.

[57] J.R. Koza, J.R. Koza i J.P. Rice. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. A Bradford book. Bradford, 1992. ISBN: 9780262111706. URL: https://books.google.pl/books?id=Bhtxo60BV0EC.

[58] Taku Kudo i John Richardson. "SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing". Sty. 2018, s. 66–71. DOI: 10.18653/v1/D18-2012.

[59] Yi-hua Lan i in. "A novel image segmentation method based on random walk". Grud. 2009, s. 207–210. DOI: 10.1109/PACIIA.2009.5406455.

[60] Fabien Lauer i Gérard Bloch. "Incorporating Prior Knowledge in Support Vector Regression". *Machine Learning* 70 (sty. 2008). DOI: 10.1007/s10994-007-5035-5.

[61] A. Lavin i S. Gray. "Fast Algorithms for Convolutional Neural Networks". *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, s. 4013–4021.

[62] Yann LeCun i in. "Efficient BackProp". *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. Berlin, Heidelberg: Springer-Verlag, 1998, s. 9–50. ISBN: 3540653112.

[63] Hao Li i in. "Pruning Filters for Efficient ConvNets". *CoRR* abs/1608.08710 (2016). arXiv: 1608.08710. URL: http://arxiv.org/abs/1608.08710.

[64] Tailin Liang i in. "Dynamic Runtime Feature Map Pruning". *CoRR* abs/1812.09922 (2018). arXiv: 1812.09922. URL: http://arxiv.org/abs/1812.09922.

[65] Zhuang Liu i in. "Rethinking the Value of Network Pruning". *CoRR* abs/1810.05270 (2018). arXiv: 1810.05270. URL: http://arxiv.org/abs/1810.05270.

[66] Minh-Thang Luong, Hieu Pham i Christopher D. Manning. "Effective Approaches to Attention-based Neural Machine Translation". *CoRR* abs/1508.04025 (2015). arXiv: 1508.04025. URL: http://arxiv.org/abs/1508.04025.

[67] Richard Lyons. *Understanding Digital Signal Processing (3rd Edition)*. Sierp. 2011. ISBN: 013702741-9.

[68] W Martin, Jens Lienig i James Cohoon. "C6.3 Island (migration) models: evolutionary algorithms based on punctuated equilibria". *Evolutionary Computation 2: Advanced Algorithms and Operators* 2 (sty. 2000).

[69] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs (3rd Ed.)* Berlin, Heidelberg: Springer-Verlag, 1996. ISBN: 3540606769.

[70] Tomas Mikolov i in. "Efficient Estimation of Word Representations in Vector Space" (2013). URL: http://arxiv.org/abs/1301.3781.

[71] B. Militzer, M. Zamparelli i D. Beule. "Evolutionary search for low autocorrelated binary sequences". *IEEE Transactions on Evolutionary Computation* 2.1 (kw. 1998), s. 34–39. ISSN: 1089-778X.

[72] Frederic Morin i Yoshua Bengio. "Hierarchical Probabilistic Neural Network Language Model". *AISTATS*. 2005.

[73] Yurii Nesterov. "A method for solving the convex programming problem with convergence rate $O(1/k^2)$". 1983.

[74] D. Osinga i K. Mack. *Deep Learning Kochbuch: Praxisrezepte für einen schnellen Einstieg*. Animals. O'Reilly, 2019. ISBN: 9783960102656. URL: https://books.google.pl/books?id=zJGHDwAAQBAJ.

[75] Jongsoo Park. i in. *Faster CNNs with Direct Sparse Convolutions and Guided Pruning*. 2016. arXiv: 1608.01409 [cs.CV].

[76] Razvan Pascanu, Tomas Mikolov i Yoshua Bengio. "On the Difficulty of Training Recurrent Neural Networks". *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. ICML'13. Atlanta, GA, USA: JMLR.org, 2013, III–1310–III–1318.

[77] Razvan Pascanu, Tomas Mikolov i Yoshua Bengio. "Understanding the exploding gradient problem". *CoRR* abs/1211.5063 (2012). arXiv: 1211.5063. URL: http://arxiv.org/abs/1211.5063.

[78] Jeffrey Pennington, Richard Socher i Christopher D. Manning. "Glove: Global vectors for word representation". *In EMNLP*. 2014.

[79] Matthew Peters i in. "Deep Contextualized Word Representations". *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, czer. 2018, s. 2227–2237. DOI: 10.18653/v1/N18-1202. URL: https://www.aclweb.org/anthology/N18-1202.

[80] M. Pietron i in. "Fast Compression and Optimization of Deep Learning Models for Natural Language Processing" (2019), s. 162–168.

[81] Marcin Pietron i Maciej Wielgosz. *Retrain or not retrain? – efficient pruning methods of deep CNN networks*. 2020. arXiv: 2002.07051 [cs.LG].

[82] Marcin Pietron i in. "Comparison of GPU and FPGA Implementation of SVM Algorithm for Fast Image Segmentation". *Proceedings of the 26th International Conference on Architecture of Computing Systems*. ARCS'13. Prague, Czech Republic: Springer-Verlag, 2013, s. 292–302. ISBN: 9783642364235. DOI: 10.1007/978-3-642-36424-2_25. URL: https://doi.org/10.1007/978-3-642-36424-2_25.

[83] Kamil Piętak i in. "Striving for performance of discrete optimisation via memetic agent-based systems in a hybrid CPU/GPU environment". *Journal of Computational Science* 31 (2019), s. 151–162. ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2019.01.007. URL: http://www.sciencedirect.com/science/article/pii/S1877750318307774.

[84] John Platt. "Fast Training of Support Vector Machines Using Sequential Minimal Optimization". *Advances in Kernel Methods: Support Vector Learning* (lut. 1999), s. 185–208.

[85] William H. Press i in. *Numerical Recipes in C: The Art of Scientific Computing*. USA: Cambridge University Press, 1988. ISBN: 052135465X.

[86] Feng Qian i in. "Unsupervised seismic facies analysis via deep convolutional autoencoders". *Geophysics* 83.3 (kw. 2018), A39–A43. ISSN: 0016-8033. DOI: 10.1190/geo2017-0524.1. eprint: https://pubs.geoscienceworld.org/geophysics/article-pdf/83/3/A39/4177220/geo-2017-0524.1.pdf. URL: https://doi.org/10.1190/geo2017-0524.1.

[87] Fletcher R. "Quadratic programming. In: Practical methods of optimization". *Advances in Kernel Methods: Support Vector Learning* (maj 2000), s. 229–258. DOI: 10.1002/9781118723203.

[88] Alec Radford, Luke Metz i Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks" (list. 2015).

[89] I. Rechenberg, B.F. Toms i Royal Aircraft Establishment. *Cybernetic Solution Path of an Experimental Problem:* Library translation / Royal Aircraft Establishment. Ministry of Aviation, 1965. URL: https://books.google.pl/books?id=FaWrtAEACAAJ.

[90] L. Rokach i O. Maimon. "Top-down induction of decision trees classifiers - a survey". *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 35.4 (2005), s. 476–487.

[91] P. Russek i in. "A custom co-processor for the discovery of low autocorrelation binary sequences". *Measurement Automation Monitoring* (2016).

[92] Hans-Paul Schwefel. *Numerical Optimization of Computer Models*. T. 33. Sty. 1981. DOI: 10.2307/2581158.

[93] Wenzhe Shi i in. "Is the deconvolution layer the same as a convolutional layer?" (Wrz. 2016).

[94] Karen Simonyan i Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". *arXiv 1409.1556* (wrz. 2014).

[95] Nitish Srivastava i in. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". *Journal of Machine Learning Research* 15 (czer. 2014), s. 1929–1958.

[96] *Strona domowa architektury VOLTA*. https://www.nvidia.com/en-us/data-center/v100/.

[97] *Strona domowa biblioteki Cublas*. https://developer.nvidia.com/cublas.

[98] *Strona domowa biblioteki cuda-math-api*. https://docs.nvidia.com/cuda/cuda-math-api/.

[99] *Strona domowa biblioteki cuDnn*. https://developer.nvidia.com/cudnn.

[100] *Strona domowa biblioteki OpenMP*. http://www.openmp.org/.

[101] *Strona domowa biblioteki TensorFlow*. https://www.tensorflow.org/.

[102] *Strona domowa języka biblioteki JCuda*. http://www.jcuda.org/.

[103] *Strona domowa języka Java*. https://www.java.com/pl/.

[104] *Strona domowa języka PYTHON*. https://www.python.org/.

[105] *Strona domowa między narodowych zawodów LSVRC*. http://www.image-net.org/challenges/LSVRC/.

[106] *Strona domowa Open AGH e-podręczniki*. https://epodreczniki.open.agh.edu.pl/.

[107] *Strona domowa platformy AgE*. https://gitlab.com/age-agh/age3/.

[108] *Strona domowa projektu REUTERS*. https://martin-thoma.com/nlp-reuters/.

[109] *Strona domowa projektu THEO-20*. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-20/www/data/.

[110] *Strona domowa środowiska CUDA*. https://developer.nvidia.com/cuda-gpus.

[111] *Strona domowa środowiska CUDA*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[112] *Strona domowa układów GPGPU z rodziny KEPLER*. https://docs.nvidia.com/cuda/kepler-tuning-guide/index.html.

[113] *Strona domowa WolframMathWorld*. https://mathworld.wolfram.com/LagrangeMultiplier.html.

[114] Aixin Sun, Ee-Peng Lim i Ying Liu. "On strategies for imbalanced text classification using SVM: A comparative study". *Decision Support Systems* 48 (grud. 2009), s. 191–201. DOI: 10.1016/j.dss.2009.07.011.

[115] Richard S. Sutton i Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[116] T. Tieleman i G. Hinton. *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning. 2012.

[117] Ashish Vaswani i in. "Attention is All you Need". *ArXiv* abs/1706.03762 (2017).

[118] Atalya Waissman i Aharon Bar-Hillel. "Input-Dependably Feature-Map Pruning". *Artificial Neural Networks and Machine Learning – ICANN 2018*. Red. Věra Kůrková i in. Cham: Springer International Publishing, 2018, s. 706–713. ISBN: 978-3-030-01418-6.

[119] Kazimierz Wiatr. *Akceleracja obliczeń w systemach wizyjnych*. Warszawa: Wydawnictwa Naukowo-Techniczne, 2003. ISBN: 8320428467.

[120] Maciej Wielgosz i Michał Karwatowski. "Mapping Neural Networks to FPGA-Based IoT Devices for Ultra-Low Latency Processing". *Sensors* 19 (lip. 2019), s. 2981. DOI: 10.3390/s19132981.

[121] Maciej Wielgosz i in. "Evaluation and Implementation of n-Gram-Based Algorithm for Fast Text Comparison". *Computing and Informatics* 36 (sty. 2017), s. 887–907. DOI: 10.4149/cai_2017_4_887.

[122] Maciej Wielgosz i in. "FPGA Implementation of the Selected Parts of the Fast Image Segmentation". *Studies in Computational Intelligence* 390 (sty. 2012), s. 203–216. DOI: 10.1007/978-3-642-24809-2_12.

[123] S. Winograd. *Arithmetic Complexity of Computations*. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial i Applied Mathematics, 1980. ISBN: 9780898711639. URL: https://books.google.pl/books?id=eWb9EF7BmCYC.

[124] Krzysztof Wróbel i in. "Comparison of SVM and Ontology-Based Text Classification Methods". Czer. 2016, s. 667–680. ISBN: 978-3-319-39377-3. DOI: 10.1007/978-3-319-39378-0_57.

[125] Krzysztof Wróbel i in. "Compression of Convolutional Neural Network for Natural Language Processing". *Computer Science* 21.1 (2020). ISSN: 2300-7036. DOI: 10.7494/csci.2020.21.1.3375. URL: https://journals.agh.edu.pl/csci/article/view/3375.

[126] Yonghui Wu i in. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". *CoRR* abs/1609.08144 (2016). arXiv: 1609.08144. URL: http://arxiv.org/abs/1609.08144.

[127] Shujian Yu i Jose Principe. "Understanding Autoencoders with Information Theoretic Concepts". *Neural Networks* (maj 2019). DOI: 10.1016/j.neunet.2019.05.003.

[128] H. Zhang i D. Li. "Naïve Bayes Text Classifier". *2007 IEEE International Conference on Granular Computing (GRC 2007)*. 2007, s. 708–708.

[129] Dominik Żurek i in. "Toward hybrid platform for evolutionary computations of hard discrete problems". *Procedia Computer Science* 108 (2017). International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland, s. 877–886. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2017.05.201. URL: http://www.sciencedirect.com/science/article/pii/S1877050917307949.

A. Załączniki

Jako dodatek do rozprawy, zostaje załączona płyta DVD, na której nagrane są implementacje powstałe na potrzeby niniejszego projektu. Płyta zawiera trzy foldery nazwane zgodnie z użytym w pracy nazewnictwem opisanych implementacji wybranych algorytmów - labs, svm, konwolucja_rzadka.

Folder *labs* zawiera cztery implementacje wykonane w środowisku CUDA, dokonujące rozwiązania problemu LABS za pomocą czterech metod:

- SDLS z lokalnością jeden - *sdls_kernel.cu*,
- Tabu serach - *tabu_kernel.cu*,
- SDLS z lokalnością 2 - *sdls_2_kernel.cu*,
- SDLS z przeszukiwaniem w głąb - *sdls_deep_through_kernel.cu*.

Folder *svm* zawiera dwie implementacje wykonane w języku C++, dokonujące kawantyzacji treningu SVM, poprzez użycie zaproponowanych w tej pracy metod. Pliki te zostały zatytułowane zgodnie z nazwami metod kwantyzujących, których używają tj. *svmMinMax.cpp* oraz *svmWithMaxMagnitiude.cpp*. Implementacje te skonfigurowane są tak by dokonać kwantyzacji do 8 bitów. Oprócz tego w folderze tym znajduje się plik o nazwie *svmGpu.cu*, który jest implementacją treningu SVM w środowisku CUDA. Program zawiera metody, które dokonują obliczeń w układach GPGPU, przy użyciu 64, 32 oraz 16 bitów. Do wspominanych implementacji załączone są pomocnicze pliki nagłówkowe *.h*, zawierające definicje niektórych użytych obiektów oraz stałych (np. ścieżki do plików wejściowych, liczbę klas obiektów itp). Dodatkowo folder zawiera skrypt wykonany w środowisku PYTHON (*svm_statistics_generator*), dzięki któremu zbierane są statystyki na temat możliwych przedziałów liczbowych, jakie mogą pojawić się w konkretnych operacjach podczas procesu treningu. Informacje te zostały wykorzystane w implementacjach przeprowadzających proces kwantyzacji.

Folder *konwolucja_rzadka* zawiera dwa foldery nazwane *16_bit* oraz *32_bit*. Tytuły te wskazują na jakim typie danych przeprowadzane są obliczenia w zawartych w nich implementacjach. Każdy z folderów zawiera trzy implementacje oznaczone zgodnie z nazwami modeli czy też typów konwolucji, którą obliczają. Plik *1x1_kernel.cu* obliczana konwoujcję 1×1, program *cnn_non_static_kernel.cu* zawiera implementację warstw sieci Cnn-non-static. Natomiast plik o nazwie *vgg_16_kernel.cu* posiada implementację warstw z modelu VGG-16.