Uniwersytet Przyrodniczy we Wrocławiu
Wydział Inżynierii Kształtowania Środowiska i Geodezji

Geodezja i Kartografia

Aleksandra Knapik

Nr indeksu 90911

Optymalizacja nadzorowanej klasyfikacji obrazów teledetekcyjnych z wykorzystaniem układów GPU i przetwarzania równoległego

Praca magisterska

Opiekun pracy

dr inż. Przemysław Tymków

Wrocław, lipiec 2015

Niniejszą pracę pragnę zadedykować mojej mamie,
która wspierała mnie nieustannie przez cały okres studiów.

Składam serdeczne podziękowania dla
Pana dr inż. Przemysława Tymków
oraz Pana mgr inż. Mateusza Karpiny
za życzliwość oraz cenne wskazówki dotyczące pracy.

Oświadczenie

1. Ja, niżej podpisany/a:
   imię (imiona) i nazwisko .....**Aleksandra Agnieszka Knapik**.....
   autor pracy dyplomowej pt.....
   .... **Optymalizacja nadzorowanej klasyfikacji obrazów teledetekcyjnych** .....
   .... **z wykorzystaniem układów GPU i przetwarzania równoległego**.....
2. Numer albumu : ...**90911**.....
3. Student/ka Wydziału .....**Inżynierii Kształtowania Środowiska i Geodezji**.....
   Uniwersytetu Przyrodniczego we Wrocławiu
4. Kierunku studiów .....**Geodezja i Kartografia**.....

Oświadczam, że ww. praca dyplomowa:

- nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz. U. Nr 24,poz.83 z poźn.zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- nie zawiera danych i informacji, które uzyskałem/łam w sposób niedozwolony.

Oświadczam również, że treść pracy dyplomowej zapisanej na przekazanym przeze mnie jednocześnie nośniku elektronicznym, jest zgodna z treścią zawartą w wydrukowanej wersji pracy, przedstawionej w procedurze dyplomowania.

Wrocław, dnia ..... 2015 roku .....

(miesiąc słownie) (czytelny podpis studenta/ki)

OŚWIADCZENIE OPIEKUNA PRACY

Oświadczam, że niniejsza praca dyplomowa została przygotowana pod moim kierunkiem w **Instytucie Geodezji i Geoinformatyki** i stwierdzam, że spełnia ona warunki do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

23.06.2015 .....

data

.....

czytelny podpis opiekuna pracy

OPTYMALIZACJA NADZOROWANEJ KLASYFIKACJI OBRAZÓW TELEDETEKCYJNYCH Z WYKORZYSTANIEM UKŁADÓW GPU I PRZETWARZANIA RÓWNOLEGŁEGO

Streszczenie pracy

Klasyfikacja obrazów teledetekcyjnych jest zadaniem wymagającym dużych nakładów mocy obliczeniowej. Współczesne mikroprocesory ogólnego zastosowania nie odznaczają się satysfakcjonującą wydajnością. W pracy podjęta została próba zredukowania czasu obliczeń klasyfikacji nadzorowanej przy użyciu procesorów kart graficznych. Wykorzystano w tym celu architekturę CUDA. Technologia ta sprawdziła się już w wielu dziedzinach, przy algorytmach o zbliżonej konstrukcji. Szczególnie ważnym aspektem pracy było odpowiednie przekształcenie standardowych rozwiązań tak, by wykorzystać cały potencjał procesora graficznego. W tym celu zamieniono sekwencyjne instrukcje na polecenia równoległe. Optymalizacji została poddana metoda najbliższego sąsiada oraz k-najbliższych sąsiadów. Czasy wykonania tych algorytmów przy zastosowaniu jedynie procesora głównego oraz CPU w połączeniu z GPU porównano ze sobą, otrzymując efekty przemawiające na korzyść drugiego podejścia. Metodę k-NN przyspieszono także na jednostce centralnej przy użyciu powszechnie stosowanej struktury k-wymiarowego drzewa. Sprawdzono następnie, która akceleracja przyniosła lepsze rezultaty. W pracy zbadano skuteczność optymalizacji obliczeń na dwóch różnych urządzeniach, dla obrazów testowych o różnej rozdzielczości.

Słowa kluczowe: teledetekcja, klasyfikacja nadzorowana, GPU, programowanie równoległe, CUDA

OPTIMIZATION OF SUPERVISED CLASSIFICATION OF REMOTE SENSING IMAGES USING GPU AND PARALLEL PROCESSING

Summary

The classification of remote sensing images is a task that requires large amounts of computing power. Current general-purpose microprocessors still do not have a satisfying performance. This thesis paper attempted to reduce computation time of supervised classification using the processor of graphics cards. Therefore CUDA architecture was used. This technology has already proven itself in many areas with algorithms of similar construction. A particularly important aspect of the study was an appropriate restatement of standard solutions to exploit the full potential of the GPU. For this purpose sequential instructions were transformed into parallel commands. The nearest neighbor and k-nearest neighbors methods endured optimalization. Execution times of these algorithms using only the central unit and the CPU in conjunction with the GPU were compared with each other. The second approach shows the most beneficial effects. K-NN method was also accelerated on the CPU using a commonly used structure of k-dimensional tree. Subsequently, it was verified which acceleration has brought better results. The study examined the effectiveness of the optimized calculations on two different devices for test images of different resolutions.

Key words: remote sensing, supervised classification, GPU, parallel programming, CUDA

Spis treści

1. Wstęp

1.1. Wprowadzenie

Klasyfikacja nadzorowana obrazów teledetekcyjnych jest zadaniem, które wiąże się z koniecznością przetworzenia pewnej ilości danych, przechowywanych w postaci obrazów – zdjęć satelitarnych bądź lotniczych. W miarę powiększania się zbioru pikseli do opracowania, zwiększa się także czas wykonywania całego procesu. Aktualnie dąży się do pozyskiwania coraz dokładniejszych zobrazowań, by rozszerzać zakres zastosowań teledetekcji, a także by polepszyć jakość produktów końcowych w dotychczasowych zadaniach wykorzystujących techniki teledetekcyjne. W tym celu nowe sensory odznaczają się coraz lepszą rozdzielczością przestrzenną, a co za tym idzie, rośnie liczba pikseli w obrazie, przypadająca na dany obszar. W ostatnich latach zaobserwować można dużą dynamikę niekorzystnych procesów, związanych ze środowiskiem naturalnym. Zjawisko to wywołane jest głównie czynnikami antropogenicznymi. Pojawia się więc potrzeba ciągłego monitoringu i oceny stanu ekosystemów. Dodatkowo, zwiększana więc jest także rozdzielczość czasowa zobrazowań. Trendy te skutkują zwyżką zapotrzebowania na moc obliczeniową procesorów, na których wykonywane jest zadanie klasyfikacji obrazów teledetekcyjnych. Aby zrealizować przedmiotowe operacje w zadowalającym czasie, stosować można rożne sposoby przyspieszenia obliczeń. Jednym z rozwiązań jest budowanie zaawansowanych algorytmów i struktur danych, innym akceleracja sprzętowa.

1.2. Cel i zakres pracy

W niniejszej pracy przedstawiona została analiza efektywności akceleracji zadania klasyfikacji nadzorowanej obrazów teledetekcyjny z wykorzystaniem procesorów kart graficznych. Motywacją do podjęcia takiego tematu były istniejące, liczne badania, które potwierdzają wydajność tego typu rozwiązań w innych algorytmach, cechujących się zbliżoną konstrukcją do klasyfikacji nadzorowanej[1][2]. Na ich podstawie można było przypuszczać, iż wykorzystanie potencjału układów graficznych, poprzez wykonanie przy ich użyciu obliczeń równoległych, może przynieść duże korzyści czasowe także w obszarze teledetekcji. Tezę tę postanowiono zbadać i zweryfikować w owym opracowaniu.

W treści pracy przedstawiono proces rozwoju akceleracji sprzętowej oraz teoretyczne podstawy prowadzenia obliczeń równoległych. W kolejnych rozdziałach omówiono także fundamentalne wiadomości na temat, wykorzystanej do badań, architektury CUDA. Główną część opracowania stanowią kolejne próby przyspieszania obliczeń przy użyciu procesora graficznego. W badaniu skupiono się na optymalizacji czasowej metod najbliższego sąsiada oraz k-najbliższych sąsiadów. Zaimplementowane one zostały zarówno w sposób klasyczny jaki i równoległy. Metodę k-NN, wykonywaną na procesorze głównym, przyspieszono dodatkowo przy użyciu struktury k-wymiarowego drzewa. Sprawdzono następnie, które z wdrożonych akceleracji przyniosły najlepsze rezultaty. Każde z rozwiązań zostało sprawdzone pod kątem poprawności wyników. Skuteczność równoległych obliczeń została przetestowana na dwóch procesorach graficznych: GeForce GT 320M oraz Tesla M2090. Badania przeprowadzono dla obrazów testowych o różnych rozdzielczościach, by sprawdzić skalowność rozwiązań równoległych.

1. GPU jako narzędzie do wykonywania zadań ogólnych

2.1. Dlaczego procesory graficzne?

Zaledwie kilkanaście lat temu programowanie równoległe przy użyciu procesorów GPU(ang. *Graphics Processing Unit*) było zajęciem uważanym za mocno specjalizacyjną gałąź informatyki, trudną w użyciu, a przez to obcą dla większości programistów. Z biegiem lat sytuacja zmieniła się diametralnie. Aktualnie, by być postrzeganym jako skuteczny programista, należy wręcz znać techniki akceleracji przetwarzania danych z użyciem kart graficznych[3]. Z czasem, ilość platform tego typu będzie z pewnością wzrastać. Dlatego, aby móc zaspokoić zapotrzebowanie społeczeństwa na nowe i bardziej wymagające produkty, należy nauczyć się z korzystać z dobrodziejstw procesorów graficznych.

2.2. Rys historyczny

2.2.1. Rozwój procesorów centralnych

Wydajność komputerów początkowo poprawiana była głównie za sprawą zwiększania szybkości taktowania zegara procesora głównego - CPU(ang. *Central Processing*

Unit). Jednakże, dalsze ulepszanie technik produkcji układów scalonych, celem zwiększania ich częstotliwości, zostało ograniczone z powodu zbyt wysokiego stopnia zużycia energii oraz właściwości fizycznych materiałów, z których są wykonywane procesory. Zaczęto więc szukać innych możliwości. Lepszą wydajność uzyskiwano następnie poprzez konstruowanie tzw. superkomputerów, tworzonych przez grupy połączonych ze sobą jednostek przetwarzających, które umożliwiały efektywną pracę w zintegrowanym środowisku. Klastry komputerowe są do tej pory wykorzystywane w instytucjach naukowych. Jednak zawsze były i nadal są rozwiązaniem niedostępnym dla przeciętnego użytkownika. W związku z tym, wzorując się na konstrukcji superkomputerów, podjęto z czasem działania mające na celu montowanie w komputerach osobistych kilku rdzeni, zamiast zaledwie jednego. W taki sposób zaczęto uzyskiwać wydajniejsze maszyny, bez konieczności zwiększania taktowania zegara CPU, czy powiększania fizycznych rozmiarów urządzenia. Stopniowo ilość dodawanych rdzeni rosła, zapewniając ciągły wzrost wydajności. To zjawisko nazywane jest rewolucją wielordzeniową i stanowi istotny krok w rozwoju komputerów osobistych.

2.2.2. Rozwój procesorów graficznych

Równolegle do ewolucji procesorów centralnych, zmianom ulegały także konstrukcje GPU. Procesor karty graficznej jest, mówiąc ogólnie, akceleratorem sprzętowym, który ukierunkowany został pierwotnie jedynie na renderowanie grafiki, czyli przetwarzanie geometrii i tekstur w kolorowe piksele, gotowe do wyświetlenia na ekranie. To urządzenie wyspecjalizowane zostało więc do wykonywania tych samych operacji matematycznych na sporej liczbie podobnych elementów. Potrzeba konstruowania takiego typu procesora pojawiła się po raz pierwszy pod koniec lat 80. zeszłego wieku. Wtedy do użytku masowego weszły systemy operacyjne posiadające graficzny interfejs użytkownika. Natomiast pierwszym procesorem graficznym, który w pewnym stopniu aproksymuje dzisiejszą konstrukcję tych urządzeń, była wprowadzona w 1999 roku karta graficzna GeForce 256 firmy NVIDIA. Był to układ, który pozwalał na jednoczesne wykonanie przekształceń i obliczeń bezpośrednio na urządzeniu. Jednakże procesory z tego pokolenia nie były w pełni programowalne - nie było możliwości konstruowania własnych metod przetwarzających. Przełomowym wydarzeniem z zakresu architektury procesorów graficznych było wydanie w 2001r. kart z serii GeForce 3. Wprowadzały one programowalne jednostki arytmetyczne o nazwie *Vertex Shader,* które służyły natywnie jedynie do cieniowania wierzchołków renderowanych obiektów. Następnie wprowadzono również jednostkę *Pixel Shader*,

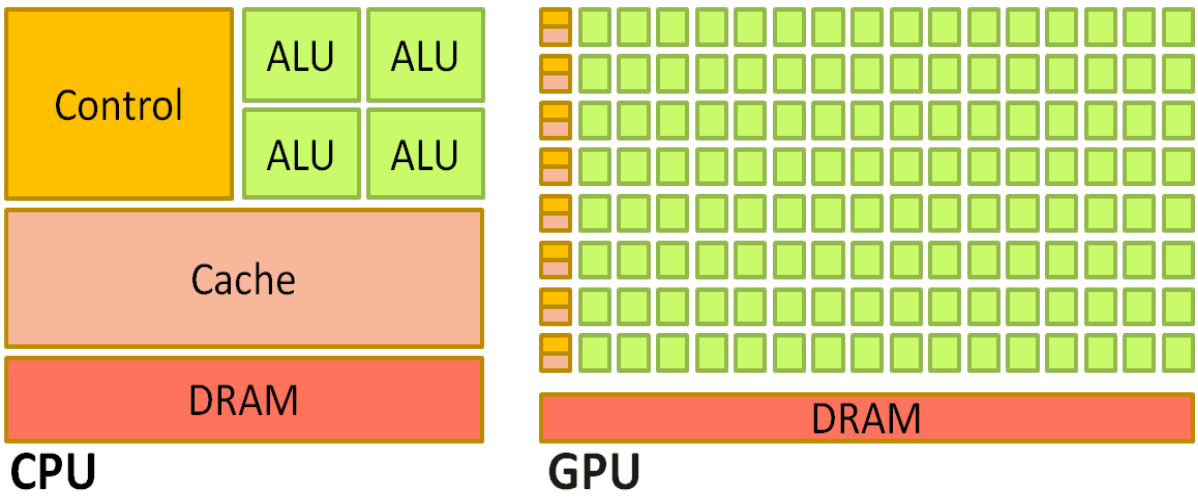
pozwalającą na sprawne obliczanie każdego z pikseli na obrazie. Wydarzenia te sprawiły, że programiści uzyskali wreszcie pełen wpływ na to, jakie dokładnie obliczenia będą mogły być realizowane na procesorze graficznym. Warunkiem interakcji z GPU, w celu wykonania zadań ogólnych (czyli innych niż związanych z grafiką), było niestety korzystanie z bibliotek graficznych DirectX bądź OpenGL. Oznaczało to dość duże trudności implementacyjne, a przez to nie wzbudzało zainteresowania większej liczby programistów. Na szczęście wkrótce nadeszły kolejne zmiany na lepsze. W listopadzie 2006r. na rynku pojawił się pierwszy GPU posiadający architekturę CUDA(ang. *Compute Unified Device Architecture*). Był to kolejny przełomowy moment w historii obliczeń ogólnych z użyciem układów

Rysunek 1 Rozwój sposobów akceleracji obliczeń
(źródło: opracowanie własne)

graficznych. Wprowadzał bowiem unifikację jednostek przetwarzających, eliminując przy tym wiele dotychczasowych ograniczeń. Przedstawiony rozwój wydarzeń przyczynił się do spopularyzowania nowego kierunku badań z zakresu inżynierii komputerowej. Powstał tzw. nurt GPGPU (ang. *General-Purpose Computing on Graphics Processing Units*), utrzymujący się do dnia dzisiejszego(*Rys. 1*). Polega on na doskonaleniu technik stosowanych do prowadzenia obliczeń nie związanych z grafiką na procesorach GPU, a co za tym idzie do ich akceleracji w jak największym stopniu. Jedną z najpopularniejszych technologii, która rozwija się wskutek tego nurtu, jest architektura CUDA, opisana szerzej w rozdziale 4 niniejszej pracy.

2.3. Komparacja GPU z CPU

Mimo, iż zarówno CPU jak i GPU zbudowane są z tych samych części funkcjonalnych, to ze względu na proporcje poszczególnych elementów(*Rys.2*), zastosowania obu typów procesorów znaczenie różnią się od siebie. Główne różnice między nimi polegają na charakterze i ilości obsługiwanych wątków, osiągalnych zasobach, jak i na sposobie dostępu do pamięci operacyjnej DRAM. Procesor główny jest zaprojektowany w sposób umożliwiający wykonanie maksymalnie szybko pojedynczego wątku, złożonego z kolejnych instrukcji. Duży nacisk kładziony jest na sprawność jednostki kontrolującej, a także zarządzanie pamięcią podręczną. Cechą charakterystyczną CPU jest szybkie, lecz jedynie



Rysunek 2 Schematyczne zilustrowanie różnic projektowych między budową CPU a GPU
(źródło: opracowanie własne na podstawie *Nvidia Programming Guide*)

sekwencyjne wykonywanie czynności przy nieskrępowanym dostępie do pamięci. Należy zwrócić uwagę na fakt, iż CPU po prostu nie zostało zaprojektowane pod kątem wykonywania obliczeń, ale w celu przetwarzania danych. Wiąże się to z małą efektywnością w przypadku realizowania skomplikowanych kalkulacji matematycznych. Procesor główny ma natomiast przewagę nad GPU jeśli chodzi o sterowanie przepływem programu i jego optymalizację. Procesor graficzny jest zaś stworzony w sposób zapewniający dużą wydajność obliczeniową. Przewagę tą zapewnia mu znacznie większa ilość jednostek arytmetyczno-logicznych ALU(ang. *Arithmetic and Logical Unit* lub *Arithmetic Logic Unit*). Tranzystory w głównej mierze zużywane są tu do wykonywania czynności obliczeniowych. W mniejszym stopniu GPU nastawione jest na sterowanie przepływem i zarządzanie pamięcią

podręczną. Procesor graficzny stanowi układ wielordzeniowy, który pozwala na równoległe zarządzanie tysiącami wątków.

2.4. Model SIMT a taksonomia Flynna

Firmy, produkujące układy graficzne przeznaczone także do wykonywania obliczeń ogólnych, takie jak NVIDIA czy AMD, definiują tego rodzaju rozwiązanie mianem architektury SIMT(ang. *Single Instruction Multiple Thread*). Taksonomia Flynna stanowi najbardziej znaną klasyfikację architektur komputerowych, zaproponowaną przez Michaela Flynna w latach 60. XX wieku. Wydziela ona cztery podstawowe typy, zróżnicowane pod względem liczby przetwarzanych strumieni danych i strumieni rozkazów. Są to następujące modele:

- SISD (ang. *single-instruction-single-data*), odpowiadająca obliczeniom w całości sekwencyjnym, gdzie przetwarzany jest jeden strumień danych przez jeden wykonywany program
- SIMD (ang. *single-instruction-multiple-data*), w którym przetwarzanych jest wiele strumieni danych, w oparciu o jeden strumień rozkazów
- MISD (ang. *multiple-instruction-single-data*), gdzie mamy do czynienia ze zwielokrotnionym strumieniem instrukcji i jednym strumieniem danych
- MIMD (ang. *multiple-instruction-multiple-data*), w którym wiele operacji jest wykonywanych na wielu zbiorach danych.

Oprócz wyżej wymienionych, głównych typów architektur, możemy de facto wyróżnić jeszcze różne ich warianty. Nie są one dodawane do powyższej systematyzacji, by nie zaciemniać ogólnego podziału modeli. Jedną z nieuwzględnianych stricte w taksonomii Flynna architektur jest właśnie SIMT. W modelu tym mamy do czynienia z jednym strumieniem rozkazów oraz wieloma wątkami, wykonującymi dane polecenie. Można się dopatrzeć się tutaj pewnej analogii z architekturą SIMD. Jednak SIMT jest o wiele łatwiejsza w użyciu, gdyż daje nam złudzenie pełnej autonomii wątków. Ponadto pozwala na wygodniejszą realizację całej logiki aplikacji.

3. Teoria obliczeń równoległych

3.1. Istota paralelizmu

Istotą wykonywania obliczeń równoległych jest realizowanie wielu zadań w tym samym czasie. Wykorzystuje się je w szczególności wtedy, gdy dany problem może zostać podzielony na mniejsze podzadania, które są od siebie niezależne, a sekwencyjne jego wykonanie wiąże się z długim czasem oczekiwania na rezultat. Stosowanie takiej formy przetwarzania danych jest zagadnieniem o wiele bardziej złożonym i trudniejszym do realizacji w praktyce niż sekwencyjne wykonywanie instrukcji. Podczas prowadzenia obliczeń w sposób równoległy możemy napotkać znacznie więcej sytuacji problemowych. Ważne jest więc bardzo dobre zapoznanie się ze specyfiką takiego podejścia, zanim przystąpi się do działania.

3.2. Warunki Bernsteina

Aby zrównoleglić wybrany algorytm, koniecznym jest wyznaczenie tzw. ścieżki krytycznej programu (ang. *Critical Path Method, CPM*). Jest to po prostu stworzenie pewnego planu, determinującego taką chronologię podzadań, by opóźnienie któregokolwiek z nich nie było przyczyną sparaliżowania całości programu. W celu znalezienia odpowiedniej ścieżki krytycznej dla obliczeń, które chcemy poprowadzić w sposób równoległy, odnajduje się najpierw wszelkie niezależne części kodu(ang. *slices*) w zaimplementowanym programie. Są to takie fragmenty, których jednoczesne uruchomienie nie spowoduje zmiany ostatecznego wyniku obliczeń. Następnie należy posegregować wybrane kawałki kodu w słusznej kolejności. To, kiedy dwa fragmenty programu P_i oraz P_j są od siebie niezależne i mogą być wykonywane w sposób równoległy bez żadnych zastrzeżeń, opisane jest Warunkami Bernsteina[4]. Mówią one, że jeśli I_i oraz I_j będą zbiorami wszystkich zmiennych wejściowych dla odpowiadających im fragmentów programu, a O_i raz O_j to zbiory zmiennych wyjściowych tych fragmentów, to części aplikacji są od siebie niezależne, gdy spełnione są poniższe warunki:

$$I_j \cap O_i = 0$$

$$I_i \cap O_j = 0$$

$$O_i \cap O_j = 0$$

Niezachowanie pierwszego bądź drugiego warunku prowadzi do tzw. zależności przepływu. Błąd ten sprawia, że wynik działania jednego fragmentu będzie wartością wykorzystywaną przez drugi fragment. Naruszenie trzeciej zależności to tzw. niezależność wyjścia, polegająca na zapisywaniu wyników operacji obydwu fragmentów kodu w tym samym miejscu, co w efekcie powoduje nadpisywanie wartości wynikowych.

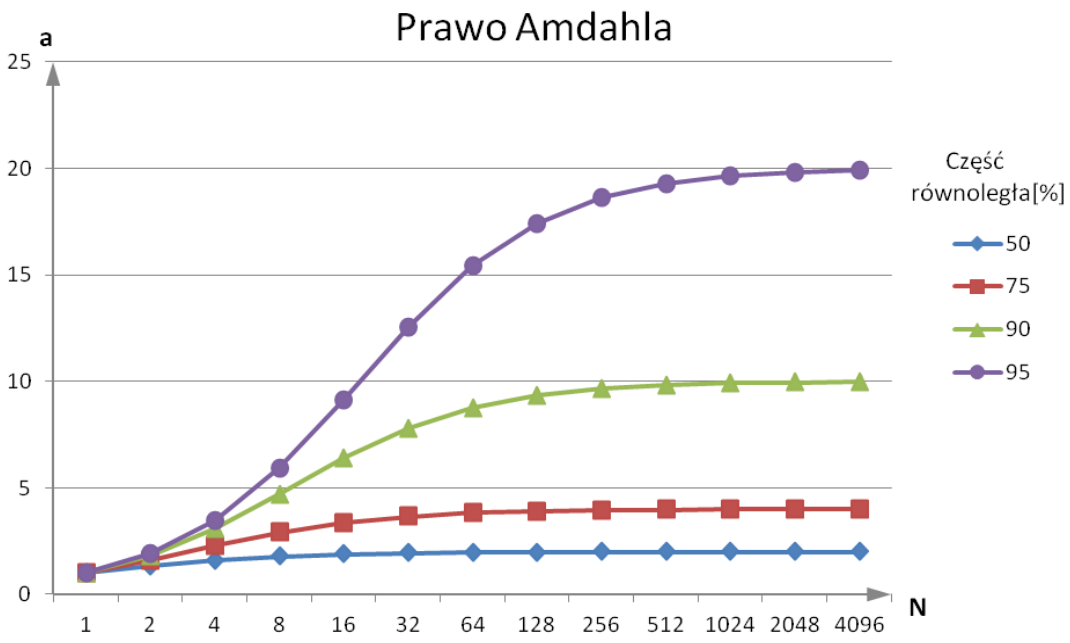
3.3. Prawo Amdahla

W celu odnalezienia maksymalnego, teoretycznego wzrostu wydajności danego algorytmu, po zastosowaniu w nim obliczeń równoległych, można skorzystać z Prawa Amdahla[5]. Mówi ono, że przyspieszenie uzyskiwane na N procesorach (w stosunku do jednego) wynosi:

$$a = \frac{1}{S + \frac{1-S}{N}}$$

gdzie: S - czas wykonania części sekwencyjnej, nie dającej się zrównoleglić

Prawo to, stworzone przez twórcę wielu architektur komputerowych – Gene Amdahla – przewiduje więc, że przy liczbie procesorów N dążącej do nieskończoności, maksymalne przyspieszenie wyniesie 1/S, ponieważ S-1/N będzie dążyło do zera. Zwiększenie mocy obliczeniowej jest w takim ujęciu ograniczone przez czas potrzebny na realizację sekwencyjnej części programu. Prawo to najprościej wyjaśnić można na przykładzie. Załóżmy, że wykonanie pewnego sekwencyjnego programu trwa 10 godzin. Przyjmijmy również, że jest on możliwy do zrównoleglenia w 90%. Wtedy czas potrzebny na wykonanie tego procesu, przy wykorzystaniu technik równoległych, będzie się zmniejszał wraz z rosnącą liczbą dostępnych procesorów. Nigdy natomiast okres ten nie będzie krótszy niż 1 godzina, czyli czas przeznaczony na niezbędne obliczenia sekwencyjne(*Wykres 1*). Należy jednak pamiętać, że wywód ten jest prawem całkowicie teoretycznym. Pomija on m.in. fakt, że obliczeń równoległych nie można idealnie podzielić między jednostki przetwarzające. Często prawo to nie jest uwzględniane podczas analiz, jeśli część sekwencyjna zajmuje niewielką część kodu.



Wykres 1 Wzrost przyspieszenia w zależności od liczby procesorów wg Prawa Amdahla (źródło: opracowanie własne)

3.4. Prawo Gustafsona

Prawo to, przedstawione w 1988r. przez Johna Gustafsona, jest ściśle związane z wyżej opisanym Prawem Amdahla. Mówi ono, że każdy wystarczająco duży problem może być efektywnie zrównoleglony[6], co zapisać możemy następująco:

$$a(P) = P - S \cdot (P - 1)$$

gdzie: a - możliwe do uzyskanie przyspieszenie
S - czas wykonania części sekwencyjnej
P - czas wykonania części równoległej

Różnica między Prawem Gustafsona, a Amdahla tkwi w podejściu do aspektu skalowalności paralelnych problemów. Pierwszy z nich zakłada możliwość ich rozbudowy, drugi nie. Ewentualne zwiększanie równoległego fragmentu programu powoduje spadek procentowego udziału części sekwencyjnej w badanym procesie, a co za tym idzie zwiększa się jego wydajność. Ideę tego twierdzenia najlepiej odzwierciedlić można przy użyciu metafory kierowcy. Zakładamy w niej, że pewien człowiek prowadzi samochód i ma do pokonania łącznie stukilometrowy odcinek drogi. Po pokonaniu pierwszych 50 km w ciągu godziny jego maksymalna średnia prędkość, jaką może osiągnąć zgodnie z teorią Amdahla,

wyniesie 100km/h. Wedle prawa Gustafsona istnieje natomiast możliwość zwiększenia wcześniej założonej odległości między punktami startowym i docelowym. W związku z tym maksymalna średnia prędkość pojazdu nie jest z góry określona.

4. Architektura CUDA

4.1. Wiadomości ogólne

CUDA(ang. *Compute Unified Device Architecture*) to stworzona przez firmę NVIDIA paralelna architektura obliczeniowa, zapewniająca znacznie lepszą wydajność w przetwarzaniu danych niż inne platformy. Przyspieszenie uzyskuje się tu poprzez wykorzystanie mocy obliczeniowej odpowiednio zaprojektowanych procesorów graficznych. W przeciwieństwie do poprzednich modeli GPU, te z architekturą CUDA nie posiadają już shaderów wierzchołków i pikseli, lecz zastosowany jest tutaj jeden, połączony potok przetwarzania[3]. Sprawia to, że aplikacje wykorzystujące tę technologię do prowadzenia obliczeń ogólnych, mogą nareszcie korzystać ze wszystkich jednostek arytmetyczno-logicznych procesora, wykonując na nich operacje zmiennoprzecinkowe. W architekturze CUDA jednostka graficzna pracuje w komputerze w roli koprocesora, asynchronicznie realizując zadania zlecone mu przez operatora. W danym systemie komputerowym może naturalnie występować więcej niż jeden procesor graficzny czy centralny. Zadaniem programisty, korzystającego z technologii CUDA, jest jak najbardziej optymalne podzielenie zakresu instrukcji pomiędzy procesor centralny i graficzny, w ten sposób, by dana jednostka realizowała zadania, w których radzi sobie najlepiej.

4.2. Wady i zalety

Aktualnie, architektura CUDA pozwala tworzyć szybko działające programy, o szerokim spektrum zastosowań. Zwiększa ich wydajność, w porównaniu z klasycznymi rozwiązaniami, nawet o kilka rzędów wielkości. Kolejną zaletą omawianej architektury jest język programowania, służący do komunikacji z urządzeniem. Stosuje się w tym celu CUDA C. Jest to rozszerzenie popularnego języka C o niezbędne do równoległego przetwarzania instrukcje, które umożliwiają kreowanie kodu źródłowego, wykonywanego po stronie GPU. Jest on znacznie łatwiejszy w użyciu niż wcześniej używane biblioteki graficzne. Ponadto,

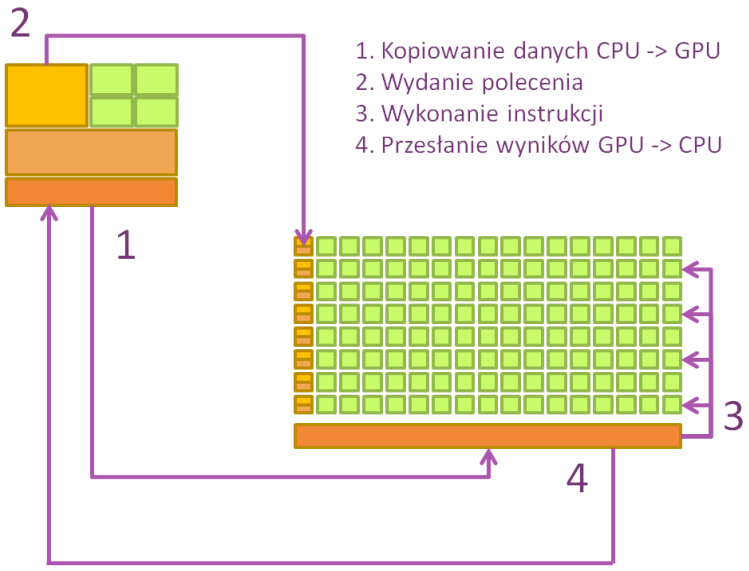
jest to rozwiązanie o znacznie lepszym stosunku jakości wykonywanych obliczeń do ceny oraz wydajności do ilości pobieranej mocy, niż ma to miejsce w przypadku prowadzenia obliczeń na CPU. Architektura ta posiada również pewne ograniczenia. Po pierwsze, będąc dziełem firmy NVIDIA, może być ona stosowana jedynie na urządzeniach tejże firmy. Z procesorów odznaczających się przedmiotową architekturą dostępne są aktualnie układy takie jak GeForce, ION, Quadro, Tesla czy Kepler. Ponadto, w tworzeniu wydajnych programów, może nam stanąć na drodze problem związany z przepustowością szyny danych miedzy CPU a GPU. Bywa ona często wąskim gardłem aplikacji.

4.3. Zastosowania

Mimo iż architektura CUDA to rozwiązanie stosunkowo młode, to jest aktualnie szeroko wykorzystywane w wielu dziedzinach. Naturalnie, technologia ta stosowana jest głównie w branży grafiki komputerowej. W szczególności popyt na takie rozwiązanie widoczny jest w sektorze związanym z tworzeniem gier komputerowych, gdzie stosowana jest ona w celu uzyskania realistycznych efektów wizualnych. Dodatkowo, platforma ta entuzjastycznie przyjęta została również na polu badań naukowych. Wykorzystano ją przykładowo w obrazowaniu medycznym. Procesory GPU z technologią CUDA przyczyniły się do stworzenia nowoczesnego systemu obrazowania ultradźwiękowego o nazwie Svara. System ten jest w stanie w ciągu zaledwie dwudziestu minut dostarczyć lekarzom bardzo szczegółowy trójwymiarowy obraz klatki piersiowej pacjenta [3]. Wcześniej tak precyzyjne badania nie mogły zostać w ogóle zrealizowane, gdyż ta metoda obrazowania wymagała zbyt dużej mocy obliczeniowej, by mogła być na co dzień wykorzystywana w diagnozowaniu pacjentów. Architektura CUDA znalazła swoje zastosowanie także w innych zagadnieniach naukowych, takich jak symulacja dynamiki płynów, kryptografia czy sztuczna inteligencja. Kompletna lista zastosowań jest wyjątkowo długa, gdyż CUDA sprawdza się wszędzie tam, gdzie mamy do czynienia z obliczeniami wielowątkowymi. Dlatego też w niniejszej pracy postanowiono sprawdzić możliwości akceleracji klasyfikacji obrazów teledetekcyjnych przy użyciu właśnie tej technologii.

4.4. Model programowania

Prowadzenie paralelnych obliczeń z użyciem CUDA odbywa się poprzez konstruowanie aplikacji działających heterogenicznie, czyli takich, które wykonują pewne instrukcje na procesorze głównym (ang. *host*), a inne na procesorze graficznym, zwanym urządzeniem (ang. *device*). Dzięki przydzielaniu części obliczeniowej zadań do jednostki GPU, która się w tym specjalizuje, dochodzi do znacznego odciążenia procesora centralnego. Przekazanie pożądanych fragmentów obliczeń na barki procesora graficznego wymaga od programisty znajomości modelu programowania równoległego(*Rys. 3*). Obliczenia z wykorzystaniem architektury CUDA rozpoczynają się klasycznie, po stronie hosta. Pierwszym etapem zrównoleglenia badanego algorytmu jest skopiowanie wszelkich niezbędnych danych, zapisanych w pamięci głównej, do pamięci urządzenia. Następnie, do procesora graficznego wysyłane są instrukcje na temat sposobu wykonania zadania. Kolejnym krokiem jest realizacja tychże instrukcji w sposób równoległy, na zadanej liczbie rdzeni. W tym miejscu należy nadmienić, iż wydanie procesorowi GPU poleceń w prawidłowy



Rysunek 3 Model programowania CUDA
(źródło: opracowanie własne na podstawie *Nvidia Programming Guide*)

sposób związane jest z wywołaniem pewnej swoistej funkcji, nazywanej jądrem(ang. *kernel*). Koniecznym jest by była ona typu void. Każdy kernel uruchamia zadaną, określoną przez programistę, liczbę wątków i wykonuje równoległe obliczenia. Tylko jedno takie jądro może być wykonywane w danej chwili. Wszystkie z uruchomionych wątków wykonują więc

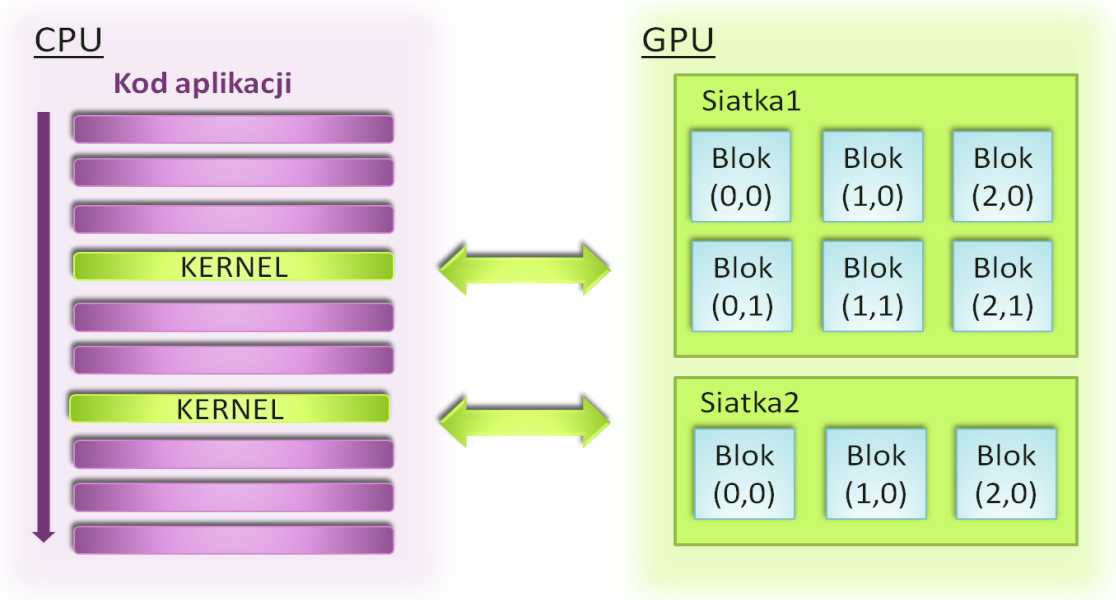
dokładnie taką samą funkcję. Każdy wątek posiada jednak identyfikator, który wykorzystywany jest do obliczania odpowiednich adresów pamięci. Kernel posiada zawsze przynajmniej jeden z trzech klasyfikatorów dostępu:

- *__global__* - który oznacza uruchomienie funkcji jądra z poziomu hosta i wykonanie jej na urządzeniu, najczęściej stosowany
- *__device__* - oznacza uruchomienie funkcji jądra z poziomu innego kernela, który jest aktualnie wykonywany na urządzeniu; wywołana funkcja wykonywana jest oczywiście po stronie GPU
- *__host__* - oznacza uruchomienie zadanej funkcji z hosta i wykonanie na hoście, używany zazwyczaj w połączeniu z __device__ do wygenerowania dwóch wersji danej funkcji

Po zakończeniu działania kernela, wyniki wykonanych operacji przesyłane są z pamięci urządzenia CUDA do pamięci procesora CPU. Kolejne etapy działania aplikacji uzależnione są od konkretnego problemu. Możemy kontynuować pracę na procesorze centralnym, uruchomić kolejny kernel na GPU, bądź zakończyć działanie programu.

4.5. Siatka, blok i wątek – organizacja

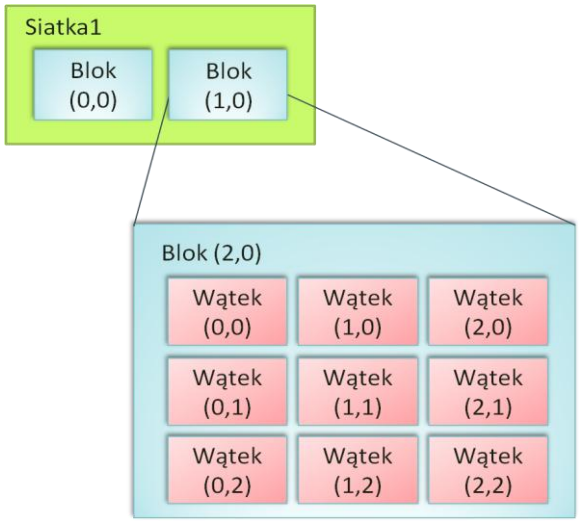
Zbiór wszystkich wątków, uruchomionych przez kernel na GPU, nazywa się siatką(ang. *grid*). Posiada ona maksymalnie dwa lub trzy wymiary, zależnie od modelu używanej karty graficznej. Siatka składa się z równoległych wywołań funkcji, nazywanych blokami (ang. *block*), które niezależnie od rodzaju urządzenia mogą być co najwyżej trójwymiarowe(*Rys. 4*). Do opisu siatki i bloku stosuje się zmienne typu dim3. Aby zidentyfikować który blok wykonuje daną kopię funkcji jądra, wykorzystuje się zmienne blockIdx.x oraz blockIdx.y, opcjonalnie także blockIdx.z. Wartości tych nie trzeba nigdzie definiować. Są to standardowe zmienne systemu wykonawczego CUDA. Zawieją one pozycję danego bloku w siatce odpowiednio w kierunku osi x, y oraz z. Tym samym określają n-wymiarowy indeks bloku, który aktualnie wykonywany jest przez urządzenie. Dzięki temu, że zdefiniowano (co najmniej) dwa wymiary, w łatwy sposób można wykonywać działania macierzowe czy też przetwarzać grafikę. Omija się w ten sposób kłopotliwe przeliczanie indeksów prostokątnych na liniowe i odwrotnie. Korzystanie z podejścia kilkuwymiarowego nie jest konieczne, jeśli dane zadanie tego nie wymaga. Do indeksacji stosować można wtedy



Rysunek 4 Podstawowy model programowania CUDA
(źródło: opracowanie własne na podstawie *Nvidia Programming Guide*)

jedynie zmienną blockIdx. Ponadto, do dyspozycji mamy także zmienne gridDim.x, gridDim.y, gridDim.z(zależne od modelu), określające wymiar całego gridu wzdłuż określonej osi. Bloki wątków, znajdujące się w danej siatce, mogą być wykonywane w dowolnej kolejności. Każdy blok z kolei składa się z wątków(*Rys. 5*), mogących współdzielić dane. Rozmiar danego bloku, czyli ilość wątków w zdefiniowanych kierunkach osi bloku, określa się poprzez zmienne blockDim.x, blockDim.y oraz blockDim.z. Wątki(ang. *threads*) mogą komunikować się miedzy sobą jednie w obrębie swojego bloku, za to przekazują informacje bardzo szybko, poprzez wspólną pamięć. Zarówno wymiar siatki jak i organizacja wątków w blokach są ustalane przez programistę przy wywoływaniu kernela. Istnieje pewne ograniczenie dotyczące ilości wątków w jednym bloku. Dla starszych układów wynosi ono 512, dla nowszych 1024. To, jakie przyjmiemy rozmiary poszczególnych elementów, ma istotny wpływ na to, jaką uzyskamy wydajność. W momencie zakończenia działania wszystkich wątków danego jądra, grid zostaje zamknięty, wyniki są przesyłane do pamięci głównej, a kierowanie programem zostaje przeniesione na stronę hosta. W trakcie programu możemy wywołać dowolną liczbę kerneli. Zaprezentowany podział pozwala na

ukrycie przed programistą znacznie bardziej złożonej budowy jednostek graficznych. Wprowadzony został on by ułatwić implementację.

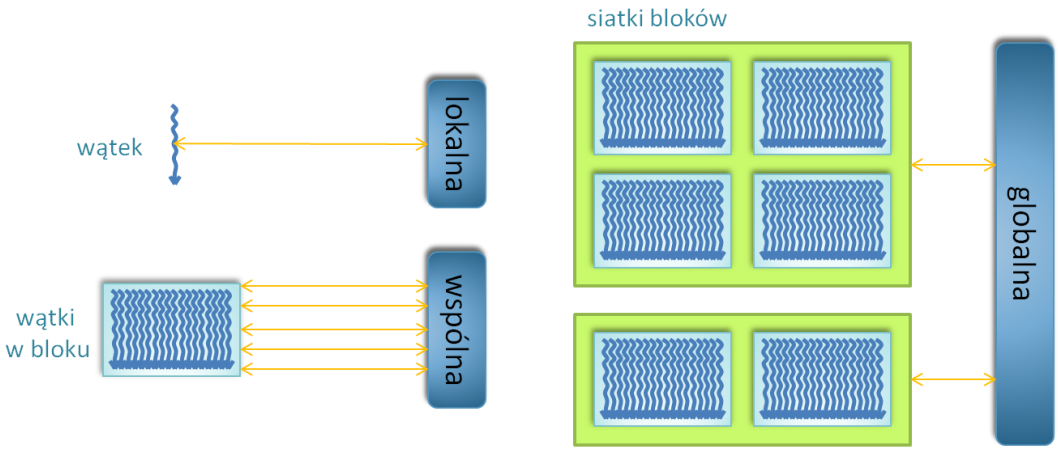


Rysunek 5 Graficzna reprezentacja organizacji wątków w bloku
(źródło: opracowanie własne na podstawie *Nvidia Programming Guide*)

4.6. Hierarchia obszarów pamięci w CUDA

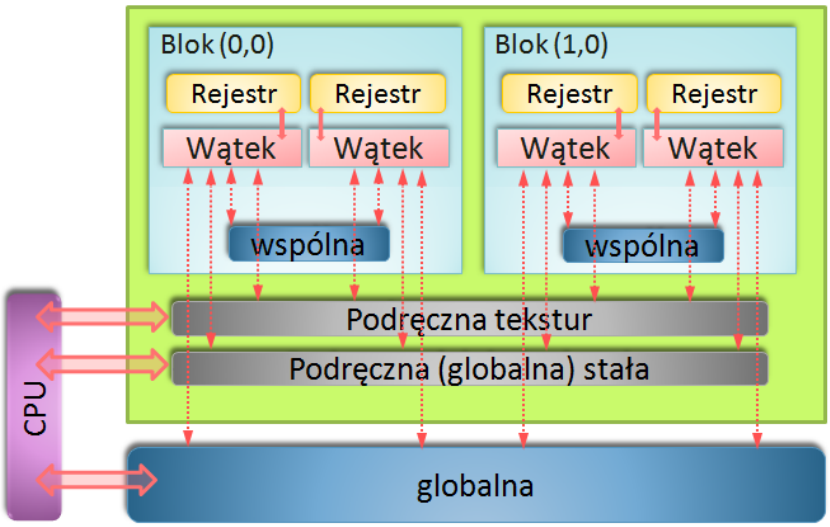
4.6.1. Podział ogólny

Wątki w opisywanej technologii mogą uzyskiwać dostęp do danych rożną drogą. Podstawowe obszary pamięci GPU hierarchizowane są w prosty, logiczny sposób(*Rys. 6*).



Rysunek 6 Hierarchia pamięci w CUDA
(źródło: opracowanie własne na podstawie *Nvidia Programming Guide*)

Każdy wątek posiada swoją osobistą pamięć lokalną. Wątki, dzielące jeden blok, mają dostęp do pamięci wspólnej. Natomiast wszystkie uruchomione wątki mogą korzystać z pamięci globalnej. Na GPU znajdziemy także dwa dodatkowe obszary tylko do odczytu. Jest to pamięć tekstur oraz stała. Obie dostępne są dla każdego z wątków[7]. W niniejszym rozdziale opisano podstawowe i towarzyszące rodzaje pamięci w CUDA(*Rys. 7*).



Rysunek 7 Związki między poszczególnymi obszarami pamięci
(źródło: opracowanie własne na podstawie *Nvidia Programming Guide*)

4.6.2. Pamięć lokalna

Pamięć ta nie jest materialnym obszarem pamięci, ale abstrakcyjnie wydzieloną częścią fizycznie istniejącej pamięci globalnej. Jest ona dedykowana dla pojedynczego wątku. Służy do przechowywania zmiennych lokalnych, które ze względu na swój rozmiar nie były w stanie zmieścić się w rejestrach. Razem z pamięcią globalną znajduje się poza układem procesora. Rozmieszczenie danych, odnoszących się pojedynczego wątku, jest niezależne od programisty. O tym, czy trafią one do rejestru czy pamięci lokalnej, decyduje kompilator.

4.6.3. Pamięć wspólna

Użycie w obliczeniach pamięci wspólnej(ang. *shared memory*) do zapisu zmiennych sprawia, że są one zupełnie inaczej traktowane. Stwarza się w ten sposób oddzielną kopię określonej zmiennej. Wszystkie wątki, w obrębie danego bloku, posiadają dostęp do kopii dla niego dedykowanej. Skorzystanie z pamięci wspólnej jest więc bardzo dobrym sposobem na komunikację i kooperację wątków dzielących ten sam blok. Plusem takiego sposobu zapisu

jest niewątpliwie fakt, iż bufory pamięci wspólnej znajdują się fizycznie na procesorze graficznym, a nie jak pamięć globalna - na odrębnym chipie. Pozwala to na redukcję czasu związanego z dostępem do pamięci. Korzystając z opisywanego rozwiązania należy koniecznie zadbać o synchronizację wątków w bloku. Trzeba bowiem zakańczać wszystkie bieżące zadania, zmieniające wartości zmiennych, zanim przystąpi się do dalszych modyfikacji. Unika się w ten sposób niepożądanej kolejności wykonywania instrukcji, a w związku z tym nieprawidłowych odczytów i zapisów nieaktualnych zawartości zmiennych. Aby zadeklarować zmienną w pamięci wspólnej wystarczy skorzystać z kwalifikatora __shared__ . Synchronizacja poszczególnych wątków w bloku odbywa się natomiast przy użyciu funkcji __syncthreads().

4.6.4. Pamięć globalna

Stanowi główny, największy obszar w pamięci kart graficznych. Jest najwolniejsza, ale za to współdzielona między wszystkie bloki w siatce, jak również między każdą z siatek. Można dokonywać jej alokacji i zwalniania, a także wykonywać zapisy i odczyty z poziomu kodu, realizowanego po stronie jednostki CPU(*Rys.7*). Jeśli istnieje zamiar wielorazowego przetwarzania tego samego fragmentu pamięci globalnej przez rożne bloki, należy pamiętać, by nie zapisywać rezultatów w miejscu czytania danych przez inne bloki. Może to bowiem doprowadzić do nieoczekiwanego działania programu. Dodatkowo, występowanie tego obszaru pamięci pozwala na zintegrowanie adresów pamięci w momencie, gdy obliczenia prowadzone są przy użyciu większej liczby procesorów GPU, skorelowanych ze sobą przy użyciu technologii SLI (ang. *Scan Line Interleave*).

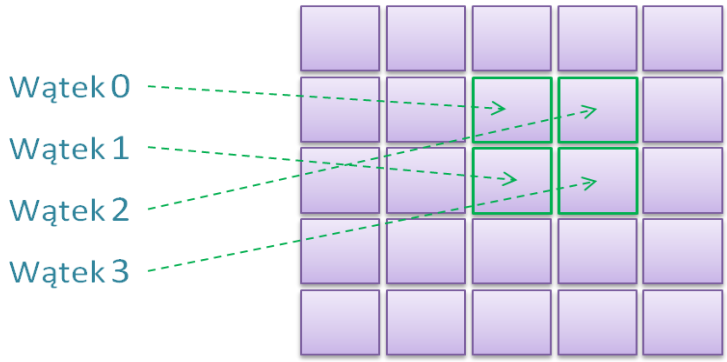
4.6.5. Pamięć stała

Procesory GPU mają nieporównywalnie więcej jednostek ALU niż CPU. W związku z tym często pojawia się kłopot z dostarczeniem im danych do przetworzenia z zadowalającą prędkością. Głównym problemem nie jest tu szybkość wykonywania obliczeń, ale przepustowość pamięci układu. Aby zredukować wykorzystanie pamięci do minimum, język CUDA C pozwala na skorzystanie z tzw. pamięci stałej(ang. *constant memory*)[8], która jest tylko do odczytu. Można w niej przechowywać wyłącznie dane nie zmieniane podczas działania funkcji jądra. Kolejnym to jej wielkość, która wynosi 64 KB. Zmniejszanie ruchu w pamięci odbywa się tutaj poprzez rozpowszechnienie pojedynczej operacji odczytu z pamięci na połowę osnowy(ang. *wrap*), czyli na zbiór szesnastu wątków. Osnowa, zwana

również paczką, jest grupą trzydziestu dwóch wątków, które są razem „splecione” i wykonywane są w sposób zsynchronizowany. Jeżeli jakaś z polówek osnowy zażąda pobrania danych z adresu, który został już wcześniej użyty, nie spowoduje to żadnego dodatkowego ruchu pamięci[3], ponieważ zawartość buforowana jest w pamięci podręcznej GPU. Taki sposób zarządzania pozwala na sporą akcelerację jedynie w przypadku gdzie wszystkie 16 wątków korzysta z tych samych danych. Gdy pobierają one jednak dane spod różnych adresów, wydajność programu się zmniejszy. Deklaracja zmiennych zapisywanych w tejże pamięci odbywa się przy użyciu modyfikatora __constant__ .

4.6.6. Pamięć tekstur

Pamięć tekstur(ang. *texture memory*) jest dodatkowym obszarem w pamięci, typowym dla GPU. Zgonie z nazwą, jej pierwotne przeznaczenie to przechowywanie danych graficznych. Jednakże możliwe jest wykorzystywanie jej również do obliczeń ogólnych. Podobnie jak w przypadku pamięci stałej, jej stosowanie może spowodować zarówno przyspieszenie jak i spowolnienie działania aplikacji. Zależy to od specyfiki rozwiązywanego problemu. Pamięć tekstur jest buforowana na układzie procesora graficznego, przez co w odpowiednich przypadkach pozwala na zmniejszenie ruchu między układem graficznym, a pamięcią DRAM (ang. *Dynamic Random Access Memory*), która znajduje się poza nim. Stosowanie tej pamięci daje zazwyczaj dobre rezultaty wtedy, gdy porcje danych do przetworzenia charakteryzują się wysokim stopniem lokalności przestrzennej[7](*Rys. 8*). Aby odczytywać dane z pamięci tekstur należy użyć funkcji tex1Dfetch(), która poinformuje GPU by żądanie było przesyłane przez jednostki teksturowe, a nie domyślnie, przez pamięć globalną.



Rysunek 8 Sposób pobierania danych przez wątki
(źródło: opracowanie własne na podstawie *Sanders J., Kandrot E. 2012*)

5. Klasyfikacja obrazów teledetekcyjnych

5.1. Teledetekcja

Podstawowym źródłem informacji o świecie w którym żyjemy są nasze zmysły. W procesie poznawania otaczającej nas przestrzeni szczególnie przydatny wydaje się być zmysł wzroku. Dzięki niemu uzyskuje się informacje o fragmentach rzeczywistości, które znajdują się w pewnej odległości od nas, nie potrzebując przy tym bezpośredniego kontaktu. Do interpretacji otoczenia ludzki receptor wzroku wykorzystuje promieniowanie elektromagnetyczne(światło), które jest odbijane od przedmiotów lub przez nie emitowane. Czopki w oku wrażliwe są jedynie na fale o określonych długościach - od 400 do 760 nm. Zakres ten nazywany jest widzialnym. Oprócz niego w widmie promieniowania elektromagnetycznego wyróżnić możemy także: fale radiowe, fale radarowe, mikrofale, podczerwień, ultrafiolet, promieniowanie X oraz promieniowanie gamma. Wymienione zakresy są możliwe do zarejestrowania przez człowieka jedynie przy użyciu specjalistycznego sprzętu. Na podstawie uzyskanych danych o intensywnościach odbicia z wielu zakresów można przeprowadzać różnego rodzaju badania i analizy. Dziedzinę nauki, która zajmuje się eksploracja właściwości obiektów lub zjawisk, uzyskując o nich informacje w sposób zdany nazywamy teledetekcją(ang. *remote sensing*). Znaczna część sensorów teledetekcyjnych, choć nie wszystkie, wykorzystuje jako nośnik informacji właśnie promieniowanie elektromagnetyczne. Rejestracja może odbywać się w niezależnych, różnych obszarach widma. Teledetekcja skupia się przede wszystkim na ustalaniu cech jakościowych badanych obiektów bądź zjawisk. Pozyskane dane gromadzone są zazwyczaj w postaci obrazów cyfrowych. Takie obrazy poddawane są kolejno różnorodnym, niezbędnym operacjom, w celu uzyskania pożądanych informacji.

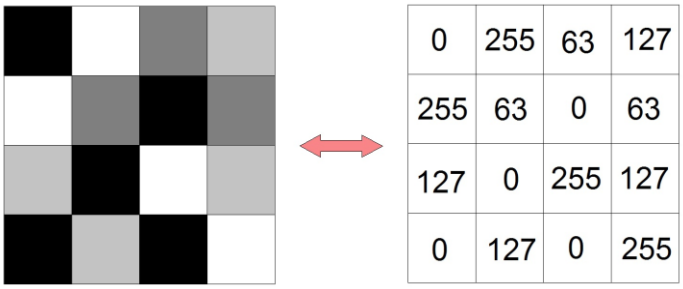
5.2. Cyfrowy obraz teledetekcyjny

Wszystkie obrazy cyfrowe przechowywane są w pamięci komputera przy użyciu jednego z dwóch podstawowych modeli: jako grafika wektorowa bądź rastrowa. Obrazy zarejestrowane przez urządzenia detekcyjne samolotu bądź satelity zapisywane są w formatach rastrowych. Powstają one poprzez przekształcenie ciągłej sceny(stanu rzeczywistego) na dyskretną tablicę skończonych elementów, którym przypisywany jest wektor cech[9]. Co można zapisać następująco:

$$L(x, y) \rightarrow L(m, n)$$

gdzie: x, y- powierzchniowe współrzędne punktu

m, n – położenie piksela w obrazie (wiersz, kolumna)



0	255	63	127
255	63	0	63
127	0	255	127
0	127	0	255

Rysunek 9 Struktura obrazu cyfrowego
(źródło: opracowanie własne)

Obraz rastrowy możemy więc identyfikować jako dwuwymiarową macierz. Aby ją uzyskać należy poddać rzeczywistą scenę procesom próbkowania oraz kwantyzacji. W formacie tym najmniejszą wyróżnialną jednostką jest piksel. Każdy z nich posiada swoją indywidualną pozycję na obrazie, określaną przy użyciu współrzędnych x oraz y. Dodatkowo, piksele posiadają też atrybuty w postaci liczb całkowitych. Dla obrazów monochromatycznych jest to zazwyczaj intensywność koloru białego(*Rys. 9*), odpowiadająca ilości zarejestrowanej energii elektromagnetycznej. Dla kolorowych wyróżniamy trzy atrybuty – natężenia barw podstawowych. Każda z tych cech przyjmuje wartości ze ściśle określonego zakresu. To, ile występuje poziomów reprezentacji, definiuje rozdzielczość radiometryczna obrazu. Zobrazowania teledetekcyjne dla danego obszaru mogą być pozyskiwane w dowolnej liczbie zakresów promieniowania elektromagnetycznego, nazywanych kanałami. Parametr, określający ich ilość to rozdzielczość spektralna.

5.3. Klasyfikacja

5.3.1. Wiadomości ogóle

Klasyfikacja obrazów teledetekcyjnych jest metodą eksploracji danych, która ma bardzo szerokie zastosowanie praktyczne. W zadaniu tym wyodrębnia się różne obiekty terenowe bądź zjawiska. Dzięki klasyfikacji możemy uzyskać wiele cennych informacji o otaczającym nas świecie. Dostarcza ona przede wszystkim szeregu aktualnych map

pokrycia terenu, wykorzystywanych m.in. w planowaniu przestrzennym, monitoringu środowiska czy zarządzaniu kryzysowym.

W praktyce rozróżnia się dwa nurty klasyfikacji – wielospektralną oraz obiektową[10]. W pierwszej podstawą jest analiza jasności pikseli w poszczególnych kanałach. W drugiej brana pod uwagę jest dodatkowo struktura i tekstura obrazu. W niniejszej pracy wykorzystano klasyfikację wielospektralną. Opiera się ona na badaniu podobieństwa odbicia spektralnego dla poszczególnych rodzajów obiektów. Podczas tego procesu przyporządkowuje się każdemu pikselowi obrazu wejściowego odpowiednią wartość, która informuje do której z klas(grup obiektów) został on przydzielony. Odbywa się to na podstawie wartości liczbowych, które odpowiadają intensywności odbicia promieniowania elektromagnetycznego w poszczególnych kanałach spektralnych. W wyniku klasyfikacji powstaje nowy obraz, składający się tylko z jednego kanału, zawierającego wartość przydzielonej klasy. Dotychczasowe praktyki wskazują, że zadanie klasyfikacji stanowi dość złożony problem. Szczególnie wtedy, gdy badany obszar odznacza się rozdrobnioną strukturą.

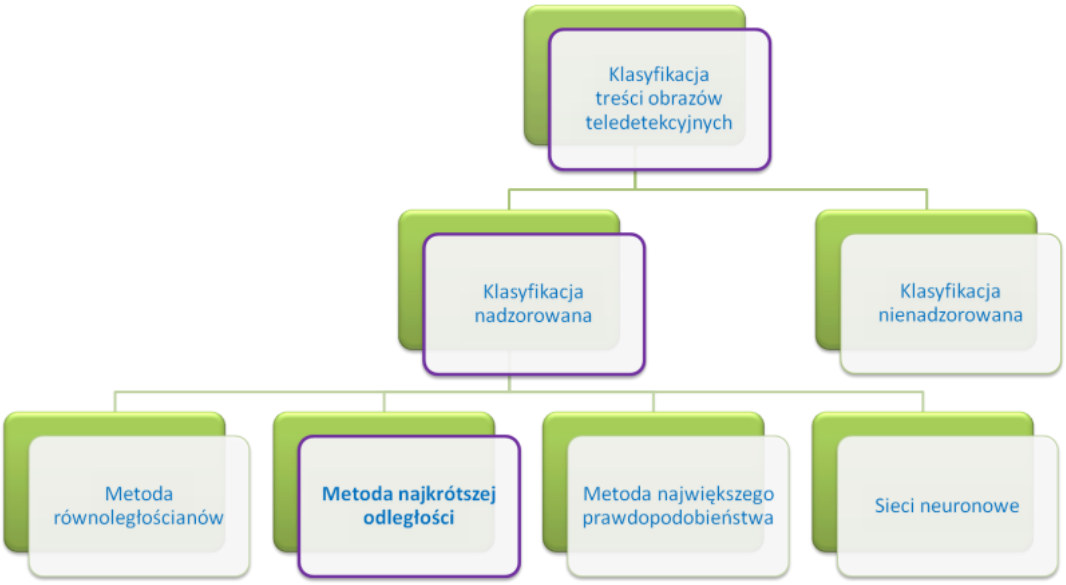
5.3.1. Podział metod

W zależności od sposobu postępowania można wyróżnić dwie główne metody klasyfikacji wielospektralnej: nienadzorowaną i nadzorowaną.

Klasyfikacja nienadzorowana(ang. *unsupervised classification*) jest procesem całkowicie zautomatyzowanym[11]. W tej metodzie system analizuje wielowymiarową przestrzeń odpowiedzi spektralnych w celu odnalezienia skupisk pikseli o zbliżonych wartościach we wskazanych kanałach. Wydzielone w ten sposób zespoły pikseli nazywane są klastrami. Następnie, poszczególnym klastrom przypisywane są odpowiednie klasy, odzwierciedlające jaki obiekt bądź zjawisko reprezentują. Aby tego dokonać wykonawca musi posiadać pewną wiedzę na temat opracowywanego obszaru, by prawidłowo zrealizować identyfikację klas, odpowiadających poszczególnym klastrom. Klasyfikacja nienadzorowana jest metodą mniej dokładną niż nadzorowana i stosunkowo mało efektywną. Stosuje się ją głównie do opracowywania obszarów niedostępnych lub traktuje jako etap przygotowawczy. Wyróżnia się między innymi następujące typy algorytmów klasyfikacji nienadzorowanej: ISODATA, statystyczna, sekwencyjna, lokalnych maksimów, klastrów RGB.

Klasyfikacja nadzorowana(ang. *supervised classification*) wymaga od operatora pewnej wiedzy a priori na temat obiektów znajdujących się na opracowywanym obszarze.

Znane obiekty wykorzystywane są jako wzorce klas, w postaci tzw. pól treningowych(ang. *training fields*). Z nimi następnie porównywany jest każdy z pikseli na obrazie. Pola te muszą być reprezentatywne dla danej klasy. Poza tym istotna jest też wielkość wzorca. Generalnie, im więcej pikseli on obejmuje, tym lepiej. Minimalna liczba pikseli pola treningowego jest określana jako dziesięciokrotność liczby kanałów obrazu cyfrowego[12]. Informacje o obiektach wzorcowych pozyskuje się przez analizę istniejących map oraz informacji tekstowych dotyczących opracowywanego obszaru, a także na podstawie zdjęć lotniczych lub wywiadu terenowego. Aby przeprowadzić klasyfikację nadzorowaną należy zastosować jeden z tzw. klasyfikatorów. Są one realizacją określonych algorytmów. Najpopularniejsze to: metoda równoległościanów, najbliższego sąsiada, największego prawdopodobieństwa oraz sieci neuronowe(*Rys. 10*). W niniejszej pracy została zbadana możliwość akceleracji algorytmów związanych z najbliższym sąsiedztwem, stąd opisano je szczegółowo w kolejnych podrozdziałach.



Rysunek 10 Podział metod klasyfikacji
(źródło: opracowanie własne)

5.3.2. Metoda NN

Metoda najbliższego sąsiada, w skrócie NN(ang. *nearest neighbour*), zwana także metodą najkrótszej odległości, jest algorytmem o stosunkowo prostej implementacji. Polega ona na odnalezieniu piksela w zbiorze uczącym(ang. *learning set*), który znajduje się najbliżej klasyfikowanego, względem wektora cech(*Rys.11*). Do klasyfikowanego piksela

przypisywana jest taka klasa, jaką ma odnaleziony piksel. Możemy zastosować tutaj różną metrykę. Na ogół stosuje się euklidesową, w której odległość wyznaczana jest wg poniższego wzoru:

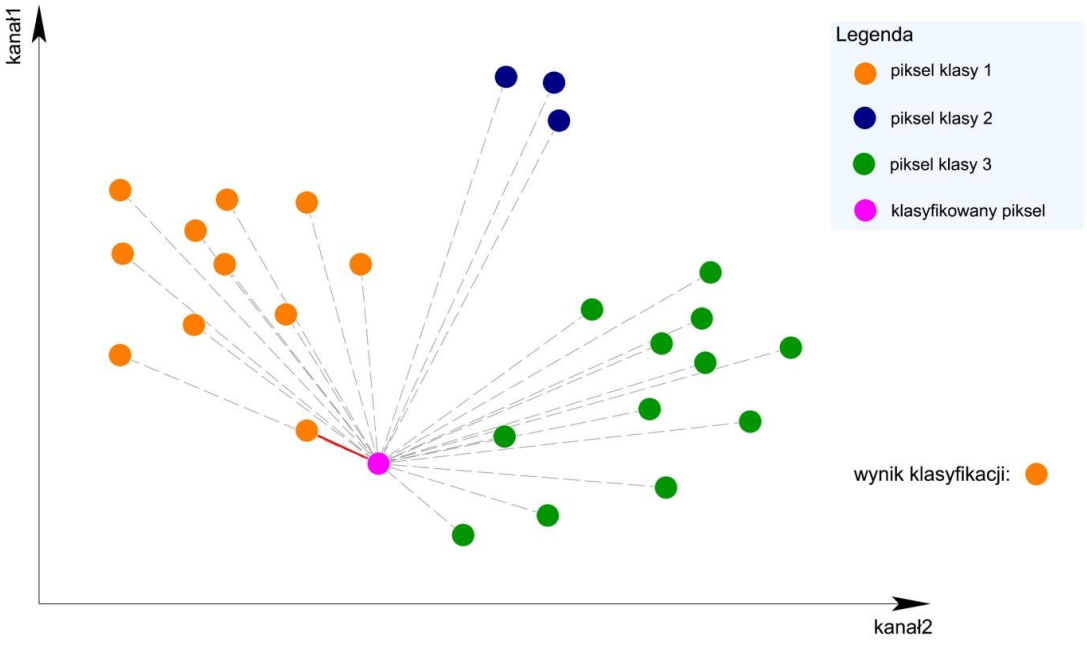
$$d^{i,j} = \sqrt{\sum_{n=1}^{k} (x_n^j - x_n^i)^2}$$

gdzie: d- odległości między j-tym pikselem ze zbioru uczącego, a i-tym ze zbioru klasyfikowanego
k – liczba kanałów spektralnych
x – wartość danej cechy

Zróżnicowana może być też ilość wymiarów przestrzeni spektralnej. Liczba ta odpowiada długości wektora cech, czyli ilości kanałów. Zmienna jest również wielkość zbioru. Przekładać się to będzie bezpośrednio na szybkość wykonywania zadania klasyfikacji. W związku z tym złożoność czasową sklasyfikowania pojedynczego obiektu(piksela) można określić następująco:

$$O(N * M)$$

gdzie: M – wymiar przestrzeni (liczba kanałów)
N - wielkość zbioru (uczącego)



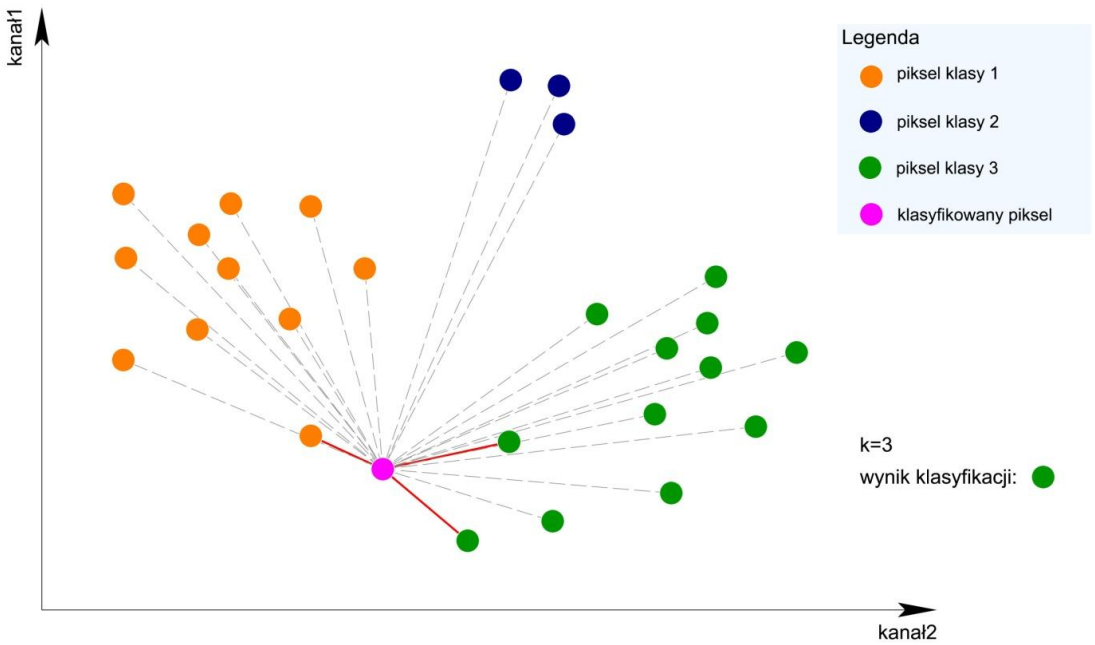
Rysunek 11 Klasyfikacja metodą NN
(źródło: opracowanie własne)

Metoda najkrótszej odległości występuje jeszcze w kilku innych wariantach. Można przykładowo nie obliczać odległości do wszystkich elementów wzorca, tylko do średniej wartości dla danej klasy. W tym celu najpierw należy odnaleźć wszystkie piksele zbioru uczącego, przynależne do danej grupy oraz wyznaczyć ich wartość średnią. Następnie zaś oblicza się odległości jedynie do tylu punktów, ile mamy klas wynikowych, a nie do tylu, ile pikseli znajduje się zbiorze uczącym. Zmniejsza to znacznie czas potrzebny na obliczenia. Taka klasyfikacja daje jednak inne, zazwyczaj gorsze rezultaty.

Sam algorytm najbliższego sąsiada stosowany może być także w różnych innych zadaniach, niezwiązanych z teledetekcją. Przykładem mogą być systemy rekomendacyjne, sekwencjonowanie DNA czy rozwijanie gier komputerowych.

5.3.3. Metoda kNN

Algorytm k-najbliższych sąsiadów, w skrócie algorytm kNN (ang. *k-nearest neighbours*) jest modyfikacją wyżej opisanego algorytmu NN. Zamiast odnajdywania jednego najbliższego sąsiada, wyszukuje się w tym przypadku kilka pikseli, znajdujących się w najmniejszej odległości. Następnie, wśród odnalezionych punktów sprawdzana jest klasa każdego z nich. Kolejno, zlicza się liczebność występowania poszczególnych klas. Klasyfikowany piksel przyporządkowywany jest do tej grupy, która występuje najczęściej

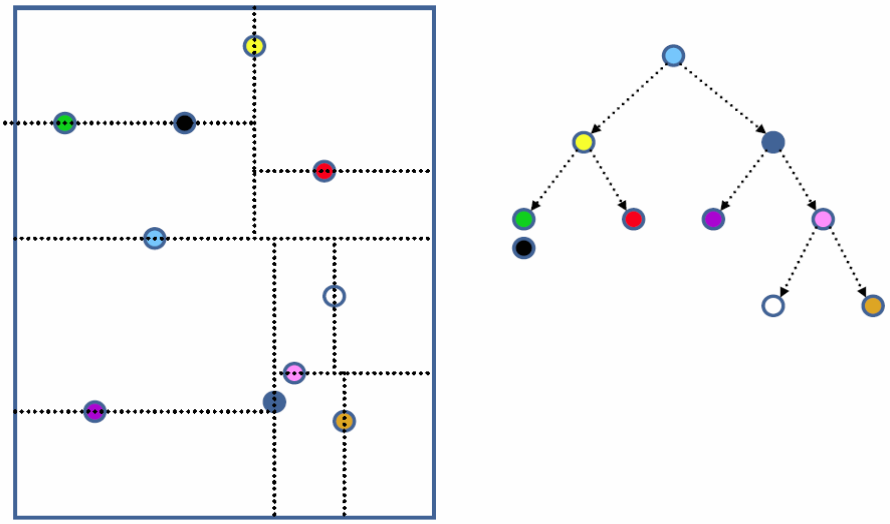


Rysunek 12 Klasyfikacja metodą kNN
(źródło: opracowanie własne)

wśród najbliższych sąsiadów(*Rys.12*). Wybór najkorzystniejszej wartości współczynnika k wymaga wykonania empirycznych prób. Nie ma uniwersalnej, ścisłej zasady, która pozwalałaby na znalezienie optymalnego k. Zaleca się jedynie, by była to wartość nieparzysta. Łatwo zauważyć, że gdy współczynnik przybiera wartość 1, algorytm zwróci nam ten sam wynik, co w przypadku zastosowania metody NN. Wzięcie pod uwagę większej liczby sąsiadów, sprawia, że algorytm ten jest bardziej odporny na szumy niż NN.

5.3.4. K-wymiarowe drzewa

Kd drzewa(ang. *k-dimensional tree, kd tree*), to binarne struktury danych, które są używane do przechowywania punktów w wielowymiarowych przestrzeniach. Służą akceleracji wykonywania algorytmów, związanych z najbliższym sąsiedztwem. Jak sama nazwa mówi, struktura ta przypomina drzewo. W zadaniu klasyfikacji obrazów teledetekcyjnych jego liście przechowują punkty ze zbioru uczącego. Każdy punkt jest przypisany jedynie do jednego liścia, a każdy liść ma przynajmniej jeden punkt(*Rys. 13*).



Rysunek 13 Zasada konstruowania i przeszukiwania struktury dwuwymiarowego drzewa
(źródło: http://wp.soulwasted.net)

Węzły drzewa odpowiadają za dzielenie przestrzeni. Podział odbywa się wzdłuż jednej z osi, rozdzielając daną przestrzeń na kolejne podprzestrzenie. Kierunek podziału może być wyznaczany naprzemiennie, po kolei dla każdej osi. Lepszym podejściem jest jednak stosowanie w tym celu odchylenia standardowego. Oblicza się je w każdym kierunku rozpatrywanej przestrzeni. Tam, gdzie uzyska się najwyższą wartość nastąpi podział. Miejsce zaczepienia węzła ustalane jest zaś tam, gdzie występuje punkt będący medianą

współrzędnych zbioru w kierunku wybranej osi. Kolejne podziały wykonywane są dopóki nieruszona pozostanie jedynie niewielka część zbioru danych (oraz przestrzeni). To kiedy część zbioru jest wystarczająco mała, by przerwać podział, określane jest przez operatora.

Struktura kd drzewa pozwala wyszukiwać zarówno jednego jak i kilku najbliższych sąsiadów. Podczas realizowania zapytania, sprawdzamy jaki liść odpowiada szukanemu punktowi. Następnie przetwarzamy punkty, które są zapisane w tym liściu, by następnie rozpocząć skanowanie pobliskich liści. W pewnym momencie, możemy zauważyć, że odległość od klasyfikowanego piksela do kolejnego elementu zbioru uczącego jest większa niż w najgorszym wypadku stwierdzonym do tej pory. W tym momencie wyszukiwanie jest zatrzymywane, gdyż kolejne liście nie poprawią wyników wyszukiwania. Taki algorytm jest skuteczny dla wyszukiwania w przestrzeniach niskowymiarowych.

Najbardziej czasochłonny krok, jaki jest wykonywany w zadaniu wyszukiwania najbliższych sąsiadów przy użyciu struktury kd drzewa, to jego budowanie. Złożoność obliczeniowa tego procesu wynosi[13]:

$$O(N \cdot logN)$$

gdzie: N - wielkość zbioru (uczącego)

Natomiast krok związany z jego jednorazowym przeszukiwaniem charakteryzuje się złożonością w przybliżeniu równą:

$$O(M \cdot logN)$$

gdzie: N - wielkość zbioru (uczącego)
M – wymiar przestrzeni(liczba kanałów)

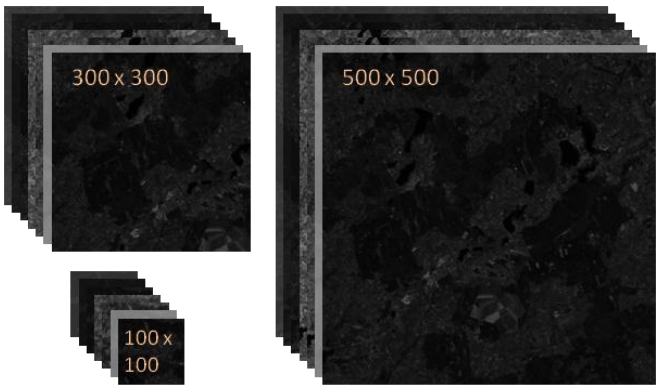
5.4. Charakterystyka danych wejściowych

5.4.1. System Landsat

Landsat jest to system teledetekcyjny, który został zaprojektowany aby dostarczać uniwersalnych informacji o powierzchni Ziemi. Prowadzony jest przez agencję NASA(ang. *National Aeronautics and Space Administration*) oraz USGS(ang. *United States Geological Survey*). Działa od 1970 roku, konsekwentnie gromadząc dane do archiwum obrazów Ziemi. Oferuje najdłuższy, ciągły zapis zobrazowań, dostarczając wizualnie i naukowo cennych

informacji o naszej planecie. Czujniki Landsat mają umiarkowaną rozdzielczość przestrzenną. Na obrazach nie jesteśmy w stanie wyróżnić mniejszych obiektów takich jak domy czy mosty. Rozdzielczość ta jest jednak wystarczająco szczegółowa by scharakteryzować procesy związane z rozwojem miast czy oceną zmiany krajobrazu w skali globalnej. W ramach programu uruchomiono do tej pory osiem satelitów[14].

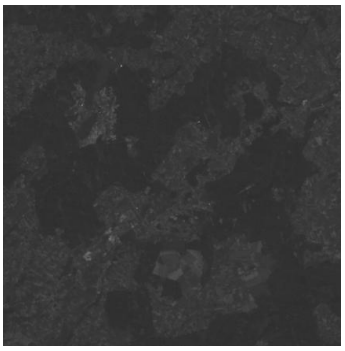
Do badań wykorzystano obrazy satelitarne pobrane ze strony USGS, pochodzące z misji Landsat, a dokładnie z sensora TM(ang. *Thematic Mapper*). Czujnik ten został wprowadzony na pokładach satelitów Landsat 4 oraz Landsat 5. Uzyskał dane obrazowe w siedmiu kanałach spektralnych, dla których rozdzielczość przestrzenna wynosi w większości 30m. Jedynie kanał szósty – termalna podczerwień, został pozyskany w rozdzielczość 120m. Jednakże poddano go resamplingowi, uzyskując taką sama rozdzielczość jak w innych kanałach. Na każdy piksel przypada 8 bitów informacji. Wydzielonych zostało więc 256 poziomów kwantowania. Przybliżony rozmiar pojedynczej sceny wynosi 170 x 185 km. Do badań wykorzystano jednak tylko fragmenty pobranych zobrazowań w trzech rozmiarach: 100x100 pikseli, 300x300 pikseli, 500x500 pikseli (*Rys. 14*). Dante te posłużyły jako zbiory uczące, a także jako zbiory do klasyfikacji.



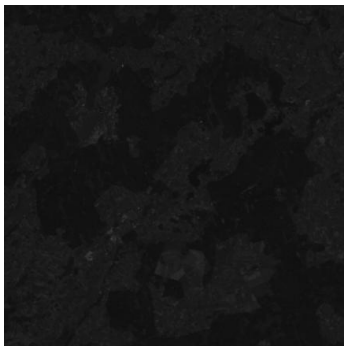
Rysunek 14 Rozdzielczości obrazów wykorzystanych w badaniach, wartości w pikselach (źródło: opracowanie własne na podstawie zobrazowań Landsat)

5.4.2. CORINE Land Cover

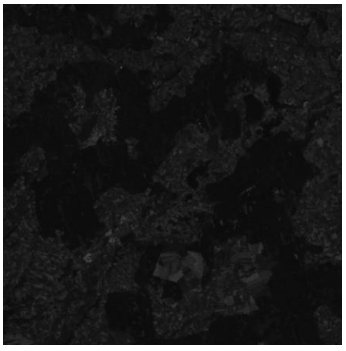
Program CORINE(ang. *Coordination of Information on the Environment*) został zainicjowany w 1985 roku przez Komisję Europejską. Jego pierwszorzędnym celem jest dokumentowanie zmian zachodzących w środowisku przyrodniczym dla krajów należących



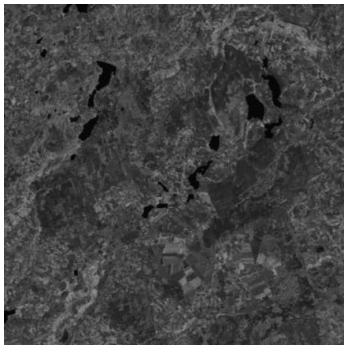
Kanał 1



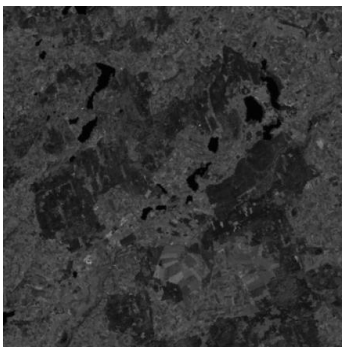
Kanał 2



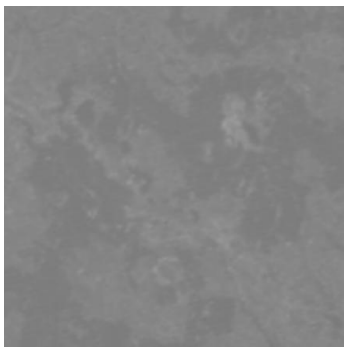
Kanał 3



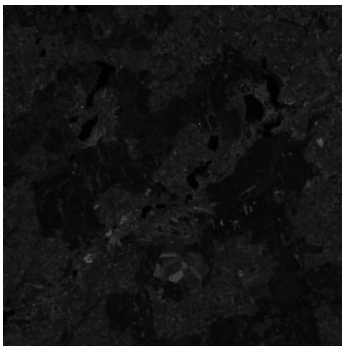
Kanał 4



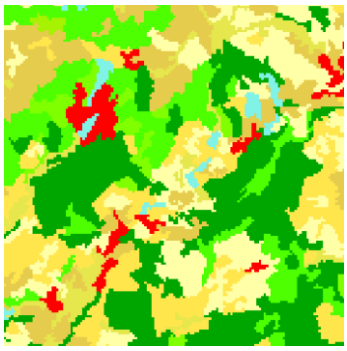
Kanał 5



Kanał 6



Kanał 7



wzorzec

Rysunek 15 Przykładowy zestaw danych uczących, wykorzystany w badaniach
(źródło: opracowanie własne na podstawie zobrazowań Landsat oraz mapy CORINE)

do Wspólnoty. Dodatkowo gromadzi się także inne dane o Europie. W Polsce program ten realizowany jest poprzez trzy sekcje tematyczne: CORINE Land Cover, CORINAIR, CORINE Biotypes. Pierwsza z nich odpowiedzialna jest za tworzenie baz danych dotyczących form pokrycia terenu. Podstawowym źródłem informacji są zdjęcia satelitarne, pochodzące z misji Landsat. Zakres oraz szczegółowość map dobrana jest do konkretnych potrzeb organów Unii Europejskiej. Ilość i rodzaj klas, występujących na mapach zależy od poziomu, w którym została ona opracowana. W pierwszym poziomie istnieje zaledwie pięć klas, w drugim piętnaście, a w trzecim rozróżnia się już czterdzieści cztery formy pokrycia terenu.

W niniejszej pracy wykorzystano wyżej opisaną CORINE Land Cover Map w celu stworzenia wzorca dla zbioru uczącego(*Rys. 15*). Kolejno, informacje o klasach zawarte na mapach CORINE, wykorzystano także jako dane weryfikacyjne do wykonania oceny dokładności klasyfikacji. Z mapy pokrycia terenu pobrano fragmenty, odpowiadające właściwym terenom z wybranych zobrazowań Landsat.

6. Przygotowanie środowiska pracy z CUDA

6.1. Procesor graficzny

Aby móc przyspieszyć działanie algorytmów przy użyciu architektury CUDA należy najpierw zaopatrzyć się w odpowiedni sprzęt. Technologia ta została stworzona przez firmę NVIDIA i tylko karty graficzne tej firmy mogą z niej korzystać. W dodatku nie wszystkie z nich posiadają odpowiednią architekturę. Niemniej jednak, jeśli karta liczy sobie mniej niż dziewięć lat, nie powinno być problemów. W razie wątpliwości, wykaz wszystkich układów, mogących skorzystać z technologii CUDA, znajduje się na stronie producenta. Istotne jest również by wybrana karta graficzna posiadała dedykowane jej, aktualne sterowniki.

Do projektu, wykonanego na potrzeby owej pracy magisterskiej, wykorzystano dwa modele kart graficznych: GeForce GT320M oraz Tesla M2090. Dzięki temu można było sprawdzić wydajność zrównoleglonych algorytmów na dwóch niezależnych urządzeniach, o różnych parametrach. Szczegółowe dane na ich temat zamieszczone są w *Tabeli 1*. Algorytmy w pełni sekwencyjne oraz niezrównoleglone fragmenty metod heterogenicznych zostały wykonane na procesorze Intel® Xeon® CPU E5649 .

Urządzenie	GeForce GT 320M	Tesla M2090
Wersja sterownika CUDA	6.0 / 6.0	7.0 / 6.5
Podrzędny numer wersji	1.2	2.0
Całkowita ilość pamięci globalnej	1024 MB	4096 MB
Ilość rdzeni CUDA	24 (3 multiprocesory po 8 rdzeni CUDA)	512 (16 multiprocesorów po 32 rdzeni CUDA
Częstotliwość zegara GPU	1100 MHz (1.10 GHz)	1301 MHz (1.30 GHz)
Częstotliwość zegara pamięci	790 MHz	1848 MHz
Szerokość magistrali pamięci	128-bit	384-bit
Maksymalny wymiar tekstury	1D=(8192), 2D=(65536, 32768), 3D=(2048, 2048, 2048)	1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)
Całkowita ilość pamięci stałej:	65536 bajtów	65536 bajtów
Całkowita ilość pamięci współdzielonej na bloku	16384 bajtów	49152 bajtów
Całkowita liczba rejestrów dostępnych na blok	16384	32768
Liczba wątków w paczce	32	32
Maksymalna liczba wątków na multiprocesor:	1024	1536
Maksymalna liczba wątków w bloku:	512	1024
Maksymalne wymiary bloku	(512, 512, 64)	(1024, 1024, 64)
Maksymalne wymiary siatki	(65535, 65535, 1)	(65535, 65535, 65535)

Tabela 1 Parametry wykorzystywanych układów graficznych
(źródło: opracowanie własne)

6.2. Narzędzia programistyczne

Fakt posiadania odpowiedniego GPU oraz sterowników jest wystarczający jedynie by uruchamiać gotowe aplikacje wykorzystujące CUDA. Jednak, aby napisać swój własny program, potrzebne jest również odpowiednie oprogramowanie. Ponieważ aplikacje, wykorzystujące procesory graficzne, należą do rozwiązań hybrydowych, niezbędne są dwa kompilatory. Pierwszy odpowiedzialny jest za kompilację kodu dla GPU, drugi dla CPU. Kompilator GPU możemy pobrać z serwisu developerskiego NVIDIA. Znajduje się on w zestawie narzędzi programistycznych o nazwie *CUDA Toolkit*. Z tego samego miejsca można dodatkowo pobrać pakiet *GPU Computing SDK*, który zawiera liczne przykłady kodu źródłowego. Kompilator dla kodu CPU, będzie natomiast zależny od systemu operacyjnego. W niniejszym badaniu korzystano z systemu Windows. W tym przypadku polecane jest użycie kompilatora Microsoft Visual Studio. Do realizacji tejże pracy została wybrana wersja 2012.

6.3. Managed CUDA

Jak wcześniej zaznaczono, językiem do komunikowania się z procesorem graficznym jest CUDA C. Natomiast język hosta może być wybrany przez programistę. W owej pracy magisterskiej zdecydowano się na użycie C#, gdyż jest on dobrze znany autorce. Aby jednak móc zintegrować kod, napisany dla hosta w tym języku, i kod urządzenia, zaimplementowany w CUDA C, należy skorzystać z wrapper'a o nazwie *Managed CUDA*. Pozwala on na łatwe połączenie przedmiotowej technologii z aplikacjami napisanymi na platformie .NET. Zapewnia on intuicyjny dostęp do sterownika *CUDA Driver API* [15]. *ManagedCUDA* jest zorientowany obiektowo. Główne klasy, wykorzystywane do implementacji kodu to:

- CudaContext – klasa ta reprezentuje kontekst CUDA. Wymagane jest bowiem stworzenie instancji przynajmniej jednego kontekstu na urządzenie. W konstruktorze można zdefiniować kilka właściwości, jak np. ID urządzenia. CudaContext określa też kilka metod statycznych, pozwalających na pobranie informacji o urządzeniach CUDA.
- CudaKernel – konstruktor, który przyjmuje trzy parametry, gdzie pierwszy to nazwa funkcji w pliku .ptx, drugi - moduł ze ścieżką do pliku i trzeci to

odpowiedni kontekst. Należy pamiętać, że nazwa metody w .ptx może różnić się od używanej w projekcie CUDA.

- CudaDeviceVariable – obiekt tej klasy reprezentuje pamięć alokowaną na urządzeniu. Klasa ta posiada informacje o dokładnym układzie pamięci GPU. Upraszcza to bardzo kopiowanie danych, gdyż nie jest potrzebne podawanie parametrów wielkości zmiennej.

6.4. Konfiguracja Visual Studio

Aby rozpocząć pracę w trybie heterogenicznym, czyli używając CPU łącznie z GPU, należy początkowo stworzyć w Visual Studio dwa projekty. Jeden powinien być typu NVIDIA CUDA, drugi w wybranym języku hosta. W przypadku tej pracy jest to język C#. Muszą one znajdować się w jednym rozwiązaniu(ang. *Solution*). Po otworzeniu automatycznie utworzonej klasy, znajdującej się w projekcie przeznaczonym dla urządzenia, będzie widoczny przykładowy kod, który można w większości usunąć. Przydatna jest jedynie funkcja jądra, poprzedzona klasyfikatorem __global__. Następnie należy umieścić pozostawiony kod kernela w nawiasach klamrowych poprzedzonych słowem kluczowym *extern ''C''* . Kolejnym krokiem jest ustawienie wymaganych parametrów dla programu CUDA. W ustawieniach ogólnych należy wybrać odpowiedni katalog wyjściowy, który stanowić powinien ścieżkę katalogu plików źródłowych projektu C#. W tej samej zakładce należy też ustawić typ konfiguracji na aplikację(.exe). Następnie w zakładce CUDA C/C++ trzeba zamienić rozszerzenie wyjścia kompilatora na .ptx oraz typ kompilacji na generujący plik .ptx. Przy takich ustawieniach dokonuje się następnie kompilacji projektu CUDA.

7. Implementacja

7.1. Zakres czynności

Próba akceleracji obliczeń przy użyciu układów GPU została przeprowadzona dla dwóch metod klasyfikacji obrazów teledetekcyjnych. Zaimplementowano algorytmy najbliższego sąsiada oraz k-najbliższych sąsiadów. Aby móc obiektywnie porównywać wydajności metod, wykonywanych sekwencyjnie i równolegle, dołożono uprzednio wszelkich starań aby jedna i druga wersja była maksymalnie zoptymalizowana. Pracę

rozpoczęto więc od znalezienia najszybszej opcji wykonania opracowywanych algorytmów przy użyciu procesora głównego. Później przystąpiono do ich implementacji i maksymalizacji wydajności przy użyciu układu graficznego. Dodatkowo, metoda kNN została zaimplementowana przy użyciu struktury k-wymiarowego drzewa, które stanowi popularny akcelerator obliczeń, związany z wyszukiwaniem najbliższych sąsiadów w przestrzeniach cech. Wszelkie uzyskane wartości, określające czas wykonania poszczególnych podejść, zestawione zostały w tabeli zbiorczej, która stanowi *Załącznik nr 1* do owej pracy. W treści rozdziału siódmego został przedstawiony przebieg badań, opis poszczególnych implementacji oraz najważniejsze zależności czasowe dla reprezentatywnych prób wraz z komentarzem. W początkowej fazie badań, opisanej w tym rozdziale, każde zadanie klasyfikacji wykonano dla siedmiu kanałów spektralnych. Natomiast w przypadku metody kNN każdorazowo brano pod uwagę 5 sąsiadów. Zachowanie jednolitych parametrów obliczeń pozwoliło na obiektywne porównanie badanych metod.

7.2. Zasady działania aplikacji

Napisany przez autorkę program jest aplikacją okienkową, która umożliwia wykonanie wyżej opisanych algorytmów w różnych wariantach(*Rys 16*). Aby rozpocząć klasyfikację wybranego obszaru należy wykonać kolejne kroki:

1. Wczytanie zbioru uczącego

   Zbiór uczący stanowi komplet dowolnej liczby obrazów w skali odcieni szarości. Każdy z nich odzwierciedla pojedynczy kanał spektralny dla wybranego obszaru. Poszczególne obrazy charakteryzować się muszą idealnie taką samą rozdzielczością oraz zakresem terenowym.
2. Wczytanie wzorca

   Należy wskazać obraz, zawierający wynik rozpoznania klas pokrycia terenu dla obszaru dokładnie odpowiadającego obszarowi ze zbioru uczącego. Na potrzeby badania skorzystano z odpowiednich fragmentów mapy CORINE Land Cover.
3. Wczytanie zbioru do klasyfikacji

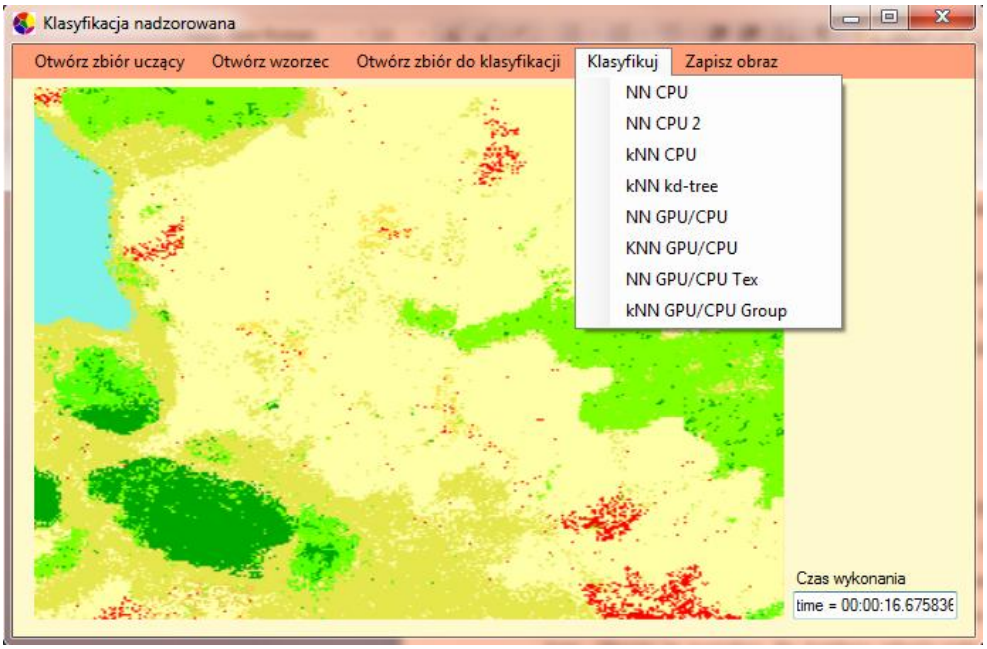
   Zbiór ten również jest zestawem obrazów, odzwierciedlających poszczególne kanały spektralne. Powinny one odpowiadać kanałom ze zbioru uczącego. Zakres terenowy każdego ze zdjęć powinien być identyczny i jest w tym przypadku jest obszarem, którego pokrycie terenu chcemy poznać.

4. Wybór sposobu wykonania obliczeń

   Dokładny opis poszczególnych metod znajduje się w kolejnych podrozdziałach. Po dokonaniu wyboru, program rozpocznie obliczenia. Wynik klasyfikacji ukaże się w postaci obrazu w panelu głównym. W prawym, dolnym rogu okienka aplikacji zostanie pokazany czas wykonania danego algorytmu.

5. Zapis wyników

   Istnieje możliwość zapisania obrazu, który przedstawia rezultat wybranej metody. Zapisane obrazy wynikowe służyć mają możliwości walidacji wyników klasyfikacji, przy użyciu w oddzielnej aplikacji, opisanej w rozdziale 8.



Rysunek 16 Okno główne programu „Klasyfikacja nadzorowana”
(źródło: opracowanie własne)

7.3. Sekwencyjny algorytm NN

Pracę rozpoczęto od wykonania metody NN na CPU, oznaczonej w kodzie jako *ClassifyImage*. Jest to podstawowa implementacja algorytmu, opisanego w rozdziale 5.3.2. Istotnym przekształceniem jest tutaj zamiana sposobu przechowywania obrazów zbioru uczącego oraz klasyfikowanego, pobranych w formie Bitmap, na dwuwymiarową tablicę typu byte. Obrazy te posiadają dla każdego piksela tylko jedną wartość z przedziału 0-255. Nie jest więc potrzebne zajmowanie tak dużego obszaru w pamięci, jaki pochłania Bitmapa. Zmiana ta pozwoliła na szybsze iterowanie po elementach obrazów podczas obliczania odległości

między poszczególnymi pikselami. Metoda pozostawała jednak wciąż bardzo nieefektywna. Zauważono, iż algorytm wykonuje się znacznie szybciej, gdy wprowadzi się w kodzie pewne zmiany. Wdrożono je w nowej metodzie o nazwie *ClassifyImage2*.

Pierwsza zmiana dotyczy procesu obliczania kolejnych dystansów między pikselami. W początkowej wersji odległość między kolejnymi ich parami obliczana była zgodnie z obowiązującym wzorem. W drugim przypadku ominięto element podnoszenia do potęgi kolejnych różnic między cechami. Ponadto, wynik dodawania różnic nie zostaje pierwiastkowany(*Listing 1*). Taki zabieg pozwala na wykonanie o jedną operację potęgowania mniej dla każdego kanału, w każdej parze pikseli, co daje $N \cdot M \cdot k$ tych zabiegów mniej. Druga modyfikacja wzoru pozwala zaś na ominięcie $N \cdot M$ operacji pierwiastkowania. W ten sposób zmieniają się co prawda wartości wyliczonych odległości, jednak nie zmienia się wynik działania klasyfikatora.

```
for (int i = 0; i < numChannels; i++)
{
    sum += (classificationSet[classificationX, classificationY, i] - learningSet[learningX, learningY, i])
        * (classificationSet[classificationX, classificationY, i] - learningSet[learningX, learningY, i]);
}
double distance = Math.Sqrt(sum);

if (distance < minDistance)
{
```

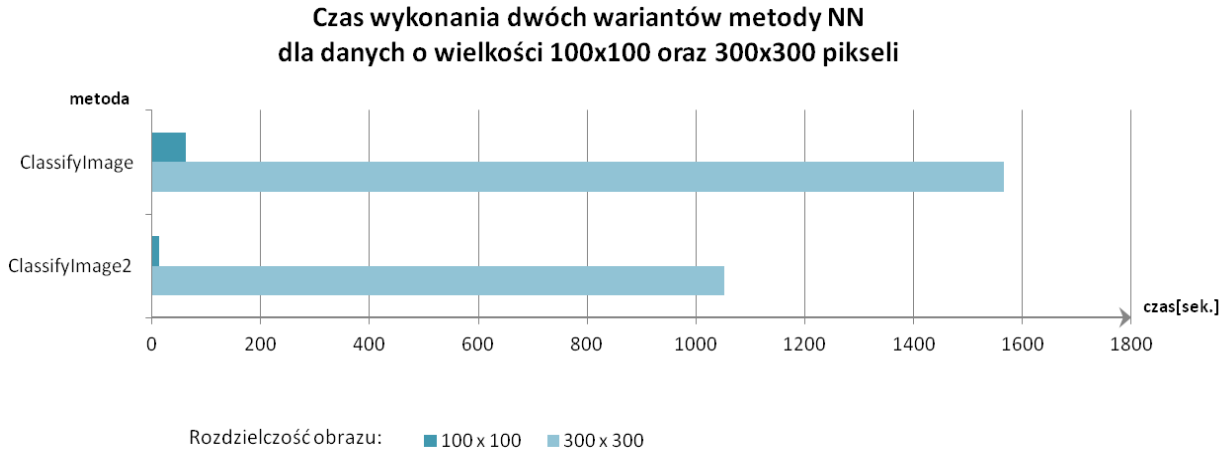
```
for (int i = 0; i < numOfChannels; i++)
{
    sum += Math.Abs(classificationSet2[classifyIdx + i * oneImageC] - learningSet2[lIdx + i * oneImageC]);
}

if (sum < minDistance)
{
```

Listing 1 Zmiana formuły obliczeniowej
(źródło: opracowanie własne)

Druga zmiana związana jest ze strukturą przechowywania zbiorów danych uczących i zbiorów do klasyfikacji. Poprzednio była to dwuwymiarowa tablica bajtów. W tym rozwiązaniu wprowadzono tablicę jednowymiarową, również typu byte. Odczyty wykonują się tutaj szybciej ze względu na sąsiedztwo lokalne kolejno pobieranych pikseli. Pamięć cache przechowuje bowiem zawsze większy zakres danych pobranych z pamięci głównej niż jest zadany. Odczytywanie fragmentów pamięci, znajdujących się w sąsiedztwie ostatnio używanych obszarów, pozwala uniknąć dodatkowych odczytów dotyczących pobliskich

wartości. Przyspieszenie uzyskane przy użyciu tych zabiegów przedstawiono na wykresie poniżej(*Wykres 2*).



Wykres 2 Rezultat optymalizacji metody NN na CPU
(źródło: opracowanie własne)

7.4. Równoległy algorytm NN

7.4.1. Metodologia

Optymalizacja czasowa algorytmów z użyciem procesora graficznego jest zadaniem znacznie trudniejszym od akceleracji na CPU i wymagającym wiedzy specjalistycznej, przedstawionej pokrótce w poprzednich rozdziałach pracy. Przede wszystkim należy mieć na uwadze czas przesyłu danych między hostem a urządzeniem. Ponadto powinno się odpowiednio gospodarować pamięcią procesora graficznego. Zrównoleglenie algorytmu rozpoczyna się zawsze od wydzielenia fragmentu kodu, który może zostać wykonany przez GPU jednocześnie dla wielu danych. W zadaniu klasyfikacji metodą NN, tym fragmentem jest główna pętla, w której wykonuje się obliczenia odległości miedzy kolejnymi pikselami[17]. Ścieżka krytyczna, uwzględniająca przeprowadzenie tych operacji równolegle, spełnia warunki Bernsteina. Oznacza to, że program powinien zwrócić takie same wyniki jak odpowiadający mu algorytm sekwencyjny, jednocześnie będąc szybszym rozwiązaniem.

7.4.2. Zrównoleglenie algorytmu

Paralelny algorytm najbliższego sąsiada rozpoczyna swoje działanie w momencie wywołania metody *cUDAToolStripMenuItem_Click*, w której początkowo wykonywane są

polecenia sekwencyjne przez CPU. W pamięci procesora głównego zapisywane są kolejno wczytywane przez użytkownika zobrazowania, z użyciem wcześniej opisanej, najkorzystniejszej struktury. Zapewnia ona najmniejszy możliwy rozmiar obrazów, co zwiększa szybkość przesyłu danych między hostem a urządzeniem. Następnie, po stronie GPU, tworzone są niezbędne kopie zmiennych, wykorzystywanych do obliczeń równoległych. W tym celu zastosowano metodę *CudaDeviceVariable*, pochodzącą z *ManagedCuda*(opis w rozdziale 6.3). W kolejnym kroku przechodzi się do kluczowego zagadnienia, jakim jest zdefiniowanie podziału wątków w obrębie bloków. W badaniu przetestowano różne wielkości bloków, by zbadać zależność między czasem wykonania zadania a ich rozmiarem(*Wykres 3*). Wymiar siatki jest natomiast ustawiony na najmniejszy możliwy przy zadanym podziale wątków i dla danej wielkości obrazów(*Listing 2*). Następnie

```
classifyWithCuda.BlockDimensions = new dim3(threadsPerBlock, 1, 1);
classifyWithCuda.GridDimensions = new dim3(
    (int)Math.Ceiling((double)(oneImageC) / classifyWithCuda.BlockDimensions.x), 1, 1);
classifyWithCuda.Run(learningSet_device.DevicePointer, classificationSet_device.DevicePointer,
    resultIdx_device.DevicePointer, oneImageL, oneImageC, numOfChannels);
```

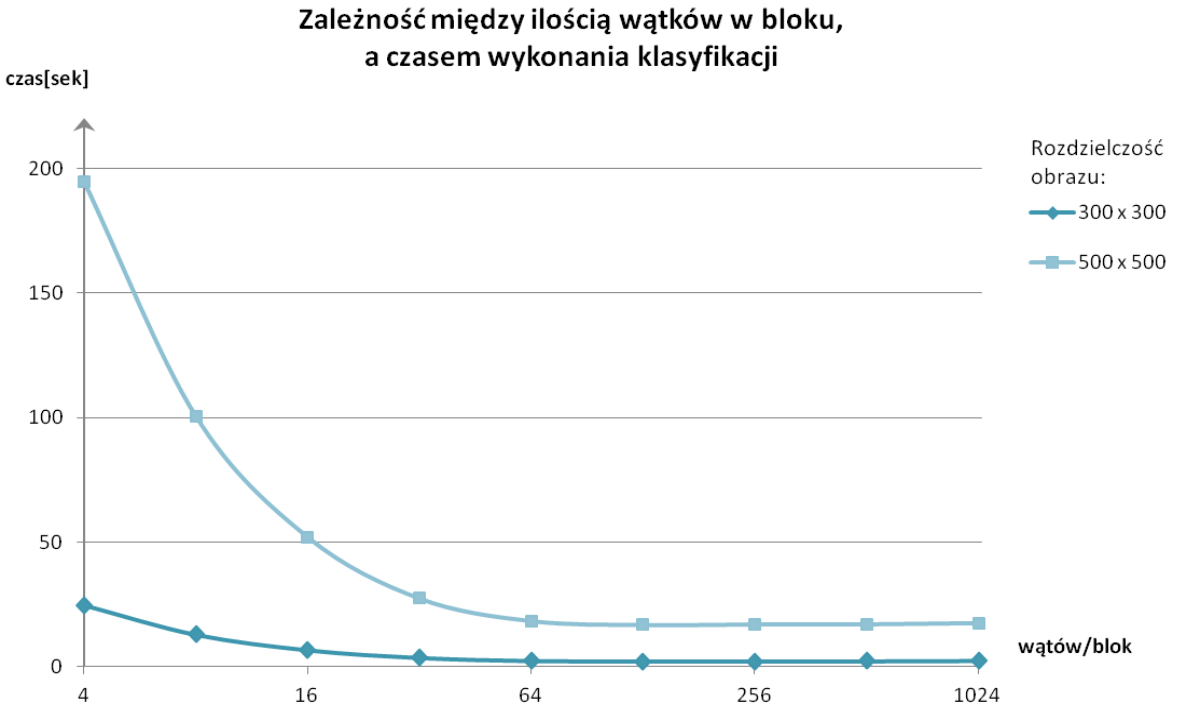
Listing 2 Organizacja wątków i wywołanie funkcji jądra
(źródło: opracowanie własne)

wywoływany est kernel *classifyWithCuda*, co powoduje, że realizacja obliczeń przeniesiona zostaje na procesor graficzny i dalsza część algorytmu wykona się tam w sposób równoległy(*Listing 2*). Każdy uruchomiony wątek odpowiedzialny jest za odnalezienie najbliższego sąsiada dla danego piksela. Stworzonych zostało więc łącznie tyle wątków, ile znajduje się pikseli w obrazie. Wynikiem działania funkcji jądra jest zbiór indeksów pikseli, odnoszących się do obrazu wzorcowego, które pokazują do jakiej klasy powinien być przypisany dany klasyfikowany piksel. Indeksy te uporządkowane są kolejno w tablicy, dzięki czemu wiadomo jakiemu pikselowi na wzorcu odpowiada dany indeks. Po zakończeniu pracy procesora graficznego, tablica z wynikami przesyłana jest do pamięci CPU. Ostatni etap obliczeń, czyli przypisanie odpowiednich kolorów stosownym pikselom na obrazie, realizowany jest po stronie CPU. Przesłanie gotowego obrazu z GPU zajęłoby bowiem więcej czasu niż sekwencyjne przypisanie kolorów po stronie hosta.

7.4.3. Analiza wydajności w zależności od podziału wątków

Jak wcześniej wspomniano, przetestowane zostały rożne sposoby rozdzielenia zadania klasyfikacji pomiędzy poszczególne bloki. Podczas badań zauważono, że im więcej

wątków umieści się w bloku, tym czas obliczeń maleje(*Wykres 3*). W pewnym momencie, zwiększanie ich ilości przestaje jednak akcelerować obliczenia. Po przekroczeniu 128 wątków na blok, nie odnotowano już przyspieszenia. Czas wykonania zadania utrzymuje się na tym samym poziomie. Sytuacja ta powtarza się zarówno dla danych o wielkości 100x100, 300x 300 pikseli, jak i dla rozdzielczości 500x500 pikseli. Przyczyną takiej sytuacji może być zbyt duża zajętość rejestrów, jednak nie da się określić jednoznacznego powodu. Nie można także uogólnić uzyskanych wyników na inne karty graficzne i algorytmy. Zależą one bowiem od stopnia skomplikowania określonego zadania oraz możliwości obliczeniowych danego GPU [16]. Najkorzystniejszy podział wątków zawsze jest najlepiej wyznaczyć w sposób empiryczny, każdorazowo przy zmianie algorytmu, urządzenia czy wielkości danych.



Wykres 3 Wyznaczanie optymalnej ilości wątków na blok
(źródło: opracowanie własne)

7.4.4. Użycie pamięci tekstur

Algorytm najbliższego sąsiada w podejściu heterogenicznym postanowiono opracować także z wykorzystaniem pamięci tekstur. Metoda, która wykonuje takie obliczenia została nazwana *classifyWithCudaTex*. Różnica między implementacją tej wersji, a kodem z klasycznym zrównolegleniem, polega na sposobie zapisania danych w pamięci GPU. Zadanie rozpoczęto od zdeklarowania zmiennych z obrazami wejściowymi, jako odwołań

```
texture<unsigned char> learningTex;
texture<unsigned char> classifyTex;
```

Listing 3 Zdeklarowanie odwołań teksturowych
(źródło: opracowanie własne)

teksturowych(*Listing 3*). Następnym krokiem było powiązanie zdeklarowanych odwołań z buforem pamięci. Wykonano tę czynność przy użyciu funkcji *BindTexture*(*Listing 4*). Ma ona za zadanie przekazanie do systemu wykonawczego informacji o planach użycia danego bufora jako tekstury oraz o zamiarze skorzystania z odwołania do danej tekstury jako jej nazwy. Gdy tekstury zostały przygotowane, można było ich użyć w obliczeniach wewnątrz kernela(*Listing 5)*. W tym celu należało skorzystać ze specjalnej funkcji *tex1Dfetch()*. Nakazuje ona przesłać żądania GPU poprzez jednostkę teksturową, a nie jak wcześniej, przez pamięć globalną.

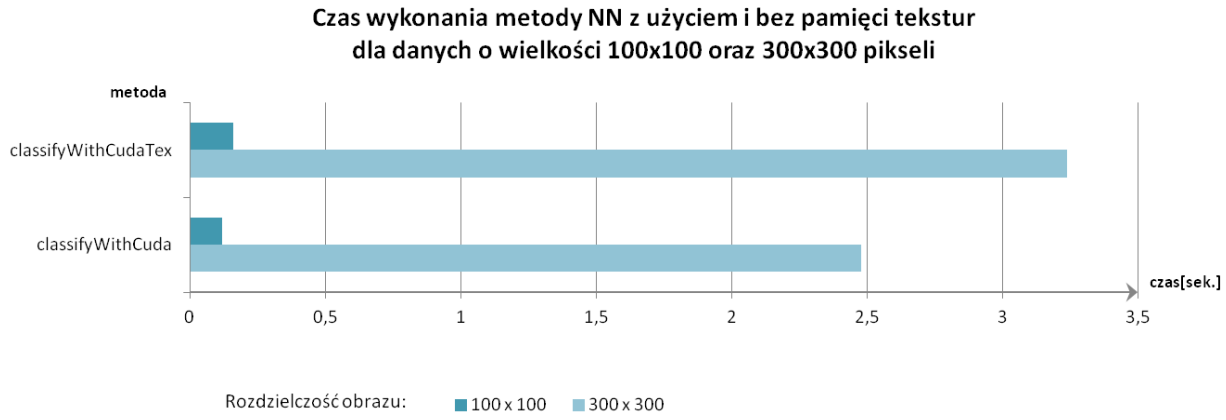
```
CudaTexture.BindTexture<byte>(classifyWithCudaTex, "learningTex", CUAddressMode.Border,
    CUTexRefSetFlags.None, CUArrayFormat.UnsignedInt8, learningSet_device);
CudaTexture.BindTexture<byte>(classifyWithCudaTex, "classifyTex", CUAddressMode.Border,
    CUTexRefSetFlags.None, CUArrayFormat.UnsignedInt8, classificationSet_device);
```

Listing 4 Powiązanie odwołań z buforem pamięci
(źródło: opracowanie własne)

```
for (int i = 0; i < numOfChannels; i++)
{
    distance +=
        fabs(((double)tex1Dfetch(classifyTex, tid * numOfChannels + i)
        - (double)tex1Dfetch(learningTex, lIdx * numOfChannels + i)));
```

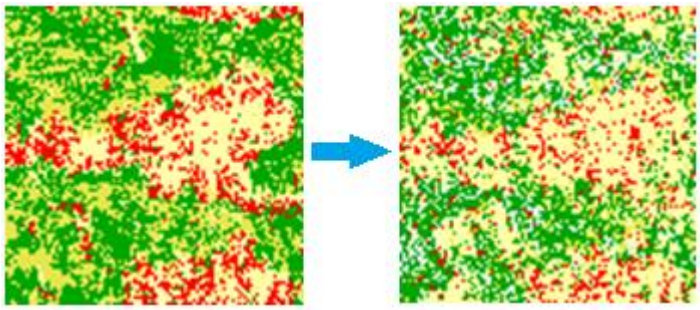
Listing 5 Użycie tekstur
(źródło: opracowanie własne)

Chociaż aplikacje napisane przy użyciu tego obszaru pamięci potrafią być znacznie wydajniejsze od ich odpowiedników, korzystających z podstawowego zestawu obszarów pamięci, w tym przypadku nie uzyskano zadowalających wyników(*Wykres 4*). Po skorzystaniu z pamięci teksturowej, wydajność programu spadła. Należało liczyć się z taką ewentualnością, gdyż skuteczność takiego podejścia zależy przede wszystkim od lokalności przestrzennej danych, do których się po kolei odwołujemy. W naszym przypadku, przy obliczaniu odległości, pobierane są każdorazowo wartości piksela w różnych kanałach, a więc znajdujące się w innej lokalizacji. Nie czerpie się danych stricte z tego samego obszaru. Dodatkowym minusem tego sposobu implementacji algorytmu jest niepoprawność obrazu



Wykres 4 Wykorzystanie pamięci tekstur w algorytmie NN
(źródło: opracowanie własne)

wyjściowego. Różni się on znacząco od prawidłowego wyniku klasyfikacji(*Rys. x*). Korzystanie z pamięci tekstur niesie więc dodatkowo ze sobą ryzyko uzyskania błędnych danych wyjściowych.



Rysunek 17 Obrazy wynikowe dla algorytmu NN na CPU oraz na GPU z użyciem pamięci tekstur
(źródło: opracowanie własne)

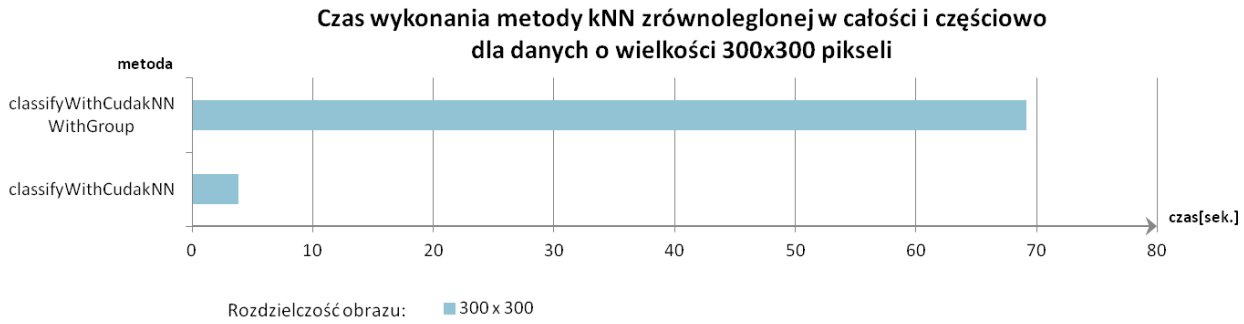
7.5. Sekwencyjny algorytm kNN

Kolejne, zbadane w tejże pracy zagadnienie to klasyfikacja obrazów teledetekcyjnych metodą k-najbliższych sąsiadów. W implementacji wykazuje ona kilka różnic w porównaniu z algorytmem wyszukującym tylko jednego, najmniej odległego piksela. Po pierwsze, każdorazowo po obliczeniu n-tej odległości między klasyfikowanym pikselem, a elementem ze zbioru uczącego, bieżąca najmniejsza wartość nie zapisywana jest w pojedynczej zmiennej lecz w k-wymiarowej tablicy. Oznacza to, że w tej metodzie nie wystarczy porównać jednej wartości, lecz należy dokonać iteracji po kilku elementach, które w danym momencie są najmniejszymi, wyznaczonymi odległościami. Poza tym, w przypadku nowej, dostatecznie małej wartości odległości należy również wiedzieć, który element tablicy

był do tej pory największy. Ta odległość powinna zostać zastąpiona nowo odnalezioną. W tym celu po każdym uaktualnieniu przedmiotowej tablicy wykonuje się jej sortowanie. Na koniec należy jeszcze dodatkowo zgrupować piksele wynikowe i zliczyć ilość elementów w poszczególnych grupach. Wszystkie te operacje powodują wydłużenie czasu obliczeń. W przypadku metody kNN, wykonywanej jedynie na hoście, nie było konieczności stosowania kolejnych etapów optymalizacji. Jako podstawę implementacji wykorzystano zoptymalizowany już algorytm NN, a następnie wprowadzono wyżej opisane modyfikacje. Przy czym od razu zachowano wszelką staranność w konstruowaniu kodu. Przedmiotowa metoda została oznaczona w kodzie jako *ClassifyImage2kNN*.

7.6. Równoległy algorytm kNN

Wyżej opisany sposób wykonywania algorytmu kNN zrównoleglono w dwojaki sposób. Metoda *classifyWithCudakNN* wykonuje po stronie procesora graficznego zadanie związane z obliczaniem odległości do poszczególnych pikseli zbioru uczącego. Następnie realizuje tutaj także proces uaktualniania i sortowania tablicy z najbliższymi sąsiadami. Po tym etapie obliczenia przenoszone są z powrotem na CPU, gdzie dokonuje się grupowania i odnalezienia wartości wynikowej. Drugi sposób zaimplementowany został jako metoda o nazwie *classifyWithCudakNNWithGroup*. Polega on na wykonaniu wszystkich wymienionych powyżej zadań przy użyciu procesora graficznego. Porównując czasy wykonania obu metod, łatwo zauważyć, że bardziej wydajny okazał się algorytm, który realizuje jedynie część obliczeń na GPU(*Wykres 5*). W przypadku obrazu o wymiarach 500x500 obliczenia z użyciem metody *classifyWithCudakNN* wykonały się ponad trzydziestokrotnie szybciej niż w przypadku metody *classifyWithCudakNNWithGroup*. Dzieje



Wykres 5 Różnice czasu wykonania klasyfikacji maetdą kNN w zależności od stopnia zrównoleglenia (źródło: opracowanie własne)

się tak dlatego, iż instrukcje realizowane w celu odnalezienia najczęściej występujących elementów w tabeli wynikowej, są zawsze wykonywane sekwencyjnie i niemożliwym jest podzielenie tego procesu na poszczególne, niezależne od siebie podzadania. Wykluczone jest paralelne wykonanie tych obliczeń. W takim wypadku lepiej więc poradził sobie procesor CPU. Jest on bowiem przystosowany do realizacji sekwencyjnych poleceń. Zaprojektowano go w sposób umożliwiający wykonanie maksymalnie szybko pojedynczego wątku, złożonego z następujący po sobie instrukcji. Układ graficzny wykonuje te obliczenia wolniej, bo nie może rozłożyć tego zadania na poszczególne multiprocesory. Nie wykorzystuje on w takim wypadku całej swojej mocy obliczeniowej.

7.7. Algorytm kNN ze strukturą kd-tree dla CPU

Akceleracja podstawowego algorytmu kNN może być realizowana nie tylko poprzez obliczenia heterogeniczne. W tym celu stosuje się również różnorakie struktury danych, pozwalające na szybsze przeszukiwanie zbioru uczącego. W niniejszej pracy postanowiono porównać wydajność algorytmu akcelerowanego przy użyciu procesora GPU z takim, który korzysta ze struktury kd drzewa. Do jego implementacji wykorzystano bibliotekę Alglib. Została ona stworzona w celu wspomagania przetwarzania danych i wykonywania różnorodnych analiz numerycznych. Między innymi posiada także metody pozwalające na efektywne budowanie k-wymiarowego drzewa oraz przeszukiwanie jego elementów.

Algorytm został napisany w sposób dedykowany procesorowi centralnemu. Struktura drzewa budowana jest poprzez zastosowanie jednego z dostępnych w bibliotece Alglib konstruktorów o nazwie kdtreebuildtagged(*Listing 6*). Tworzy on pożądane połączenia na

```
alglib.kdtreebuildtagged(learningSet3, tags, numOfChannels, 0, 2, out kdt);
```

Listing 6 Budowa struktury kd drzewa
(źródło: opracowanie własne)

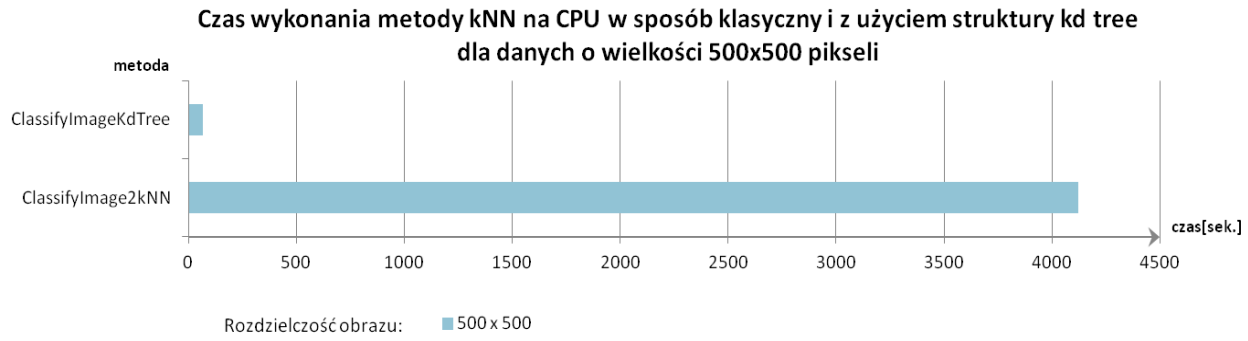
podstawie parametrów takich jak: zbiór danych, do których porównuje się klasyfikowany element, tabela z indeksami, liczba cech dla pojedynczego elementu zbioru, wiadomość czy chcemy dołączyć dodatkową wartość Y oraz rodzaj przestrzeni metrycznej. Przeszukiwanie utworzonej struktury odbywa się poprzez zastosowanie sekwencji funkcji, które pozwalają wyłuskać informacje o najbliższym sąsiedztwie danego punktu(*Listing* 7). Odpowiadają one

kolejno za budowę zapytania, zapis wartości jasności najbliższych sąsiadów, określenie ich indeksów oraz odległości od klasyfikowanego punktu.

```
int k = alglib.kdtreequeryknn(kdt, classifiedPixel, nrOfNg);
alglib.kdtreequeryresultsx(kdt, ref r0);
alglib.kdtreequeryresultstags(kdt, ref r1);
alglib.kdtreequeryresultsdistances(kdt, ref r2);
```

Listing 7 Korzystanie ze struktury kd drzewa
(źródło: opracowanie własne)

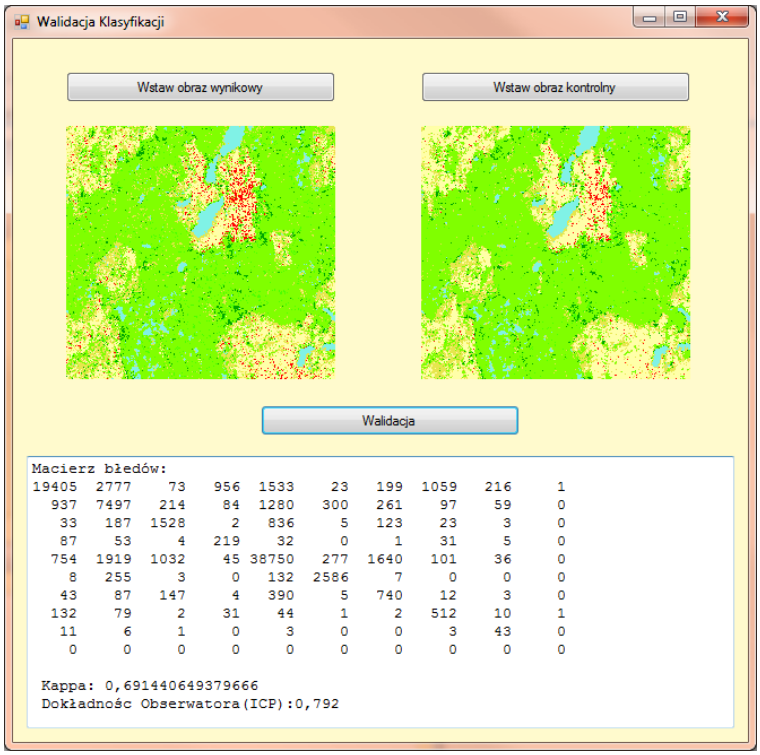
Koncept przyspieszenia klasyfikacji, polegający na wspomożeniu się strukturą kd drzewa, okazał się być bardzo skutecznym rozwiązaniem w porównaniu do klasycznego podejścia na procesorze CPU. Przyspieszenie rośnie wraz ze wzrostem zbiorów danych. Dla obrazów o największej z badanych rozdzielczości, czas wykonania zadania zmniejszył się ponad sześćdziesięciokrotnie(*Wykres 6*).



Wykres 6 Akceleracja obliczeń przy użyciu struktury kd drzewa
(źródło: opracowanie własne)

8. Walidacja

Każdorazowo, po zaimplementowaniu kolejnej metody, sprawdzano jej poprawność korzystając ze standardowych wskaźników do weryfikacji wyników klasyfikacji. Testowanie czy dany algorytm jest prawidłowy odbywało się przy użyciu oddzielnej, autorskiej aplikacji. Program *WalidacjaKlasyfikacji* porównuje dwa obrazy wczytane przez użytkownika, a następnie wylicza najważniejsze, z perspektywy wykonywanych w niniejszej pracy badań, wskaźniki dokładnościowe.



Rysunek 18 Okno aplikacji „Walidacja Klasyfikacji”
(źródło: opracowanie własne)

Macierz błędów(ang. *error matrix, confusion matrix*) stanowi fundament oceny ilościowej klasyfikacji. Określa do jakiej klasy „j” zostały przyporządkowane piksele z klasy „i”. Na jej podstawie możliwe jest wyliczenie różnych parametrów dokładnościowych.

Współczynnik Kappa przyjmuje zawsze wartości z zakresu 0-1. Definiuje on stopnień podobieństwa uzyskanych klas z obszarem referencyjnym, Im jest wyższy tym większy stopień zgodności między polem testowym, a wynikiem klasyfikacji został uzyskany. Współczynnik Kappa nazywany jest także łącznym błędem klasyfikacji. Obliczany jest on na podstawie następującego wzoru:

$$Kappa = \frac{p_0 - p_e}{1 - p_e}$$

gdzie: p_0 *– zaobserwowany poziom zgodności*

p_e *– zgodność oczekiwana*

Nie istnieje znormalizowana interpretacja współczynnika Kappa. W literaturze odnaleźć można różne pozycje, w których granice poszczególnych poziomów współczynnika wyglądają za każdym razem odmiennie. Najczęściej zakłada się jednak następujący podział:

Słaba zgodność:	0,00 – 0,20
Przeciętna zgodność:	0,21 – 0,40
Dobra zgodność:	0,41 – 0,60
Istotna zgodność:	0,61 – 0,80
Prawie idealna zgodność:	0,81 – 1,00

Dokładność obserwatora jest wskaźnikiem, który mówi jaki jest udział pikseli sklasyfikowanych poprawnie w stosunku do wszystkich pikseli na obrazie, co ująć można przy pomocy wzoru:

$$I_{CP} = \frac{\sum_{p=1}^{k} n_p}{n}$$

gdzie: k *– ilość klas*

n_p *– piksele sklasyfikowane poprawnie dla danej klasy*

n *– ilość wszystkich pikseli na obrazie*

Walidację rozpoczęto od sprawdzenia dokładności podstawowych metod NN oraz kNN, wykonywanych na CPU. W tym celu do programu wczytano najpierw obraz wynikowy metody *ClassifyImage2* oraz fragment mapy CORINE Land Cover, odpowiadający terenowi ze zbioru klasyfikowanego. Walidację przeprowadzono dla danych o rozmiarze 300 x 300 pikseli, otrzymując następujące wyniki:

$$Kappa = 0{,}35$$

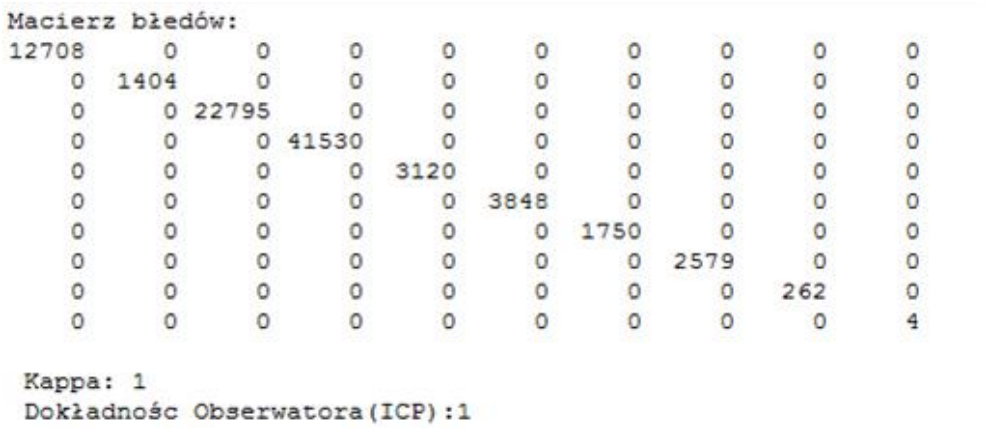
$$I_{CP} = 0{,}56$$

Uzyskany rezultat nie należy do najlepszych, ale można go w tym przypadku uznać za zadowalający. Dane do testów nie były bowiem dobierane w celu uzyskania jak najbardziej wiarygodnych wyników. Służyły one jedynie do możliwości przeprowadzenia badań związanych z optymalizacją czasową zadania klasyfikacji. Następnie, dla tych samych danych, zbadano też dokładność algorytmu kNN, uzyskując nieco lepsze wyniki, które prezentują się następująco:

$$Kappa = 0{,}41$$

$$I_{CP} = 0{,}62$$

Kolejno, przeprowadzono najważniejsze testy. Mianowicie sprawdzono czy przenosząc część obliczeń na procesor graficzny uzyska się ten sam efekt, co w przypadku obliczeń na CPU. Aby móc stwierdzić, że użycie procesora graficznego nie wpływa na rezultat klasyfikacji, wynikowe obrazy powinny być całkowicie zgodne. Oznacza to, iż współczynnik Kappa oraz dokładność obserwatora powinny posiadać wartość równą jeden. Macierz błędów powinna zaś posiadać poza przekątną jedynie wartości równe zeru(*Rys. 19*).



Rysunek 19 Przykładowy wynik walidacji
(źródło: opracowanie własne)

Walidację metod związanych z GPU wykonano na podstawie danych o rozmiarze 300x300 pikseli. Wyniki zestawiono w *Tabeli 2* . Większość implementacji dostarczyła poprawnych wyników. Jedynie obliczenia z wykorzystaniem pamięci tekstur zmieniły rezultat klasyfikacji. Wykorzystywanie tego obszaru pamięci spowodowało bardzo duże różnice w obrazie wyjściowym. Metoda ta nie nadaje się więc zdecydowanie do wykonywania zadania klasyfikacji obrazów teledetekcyjnych.

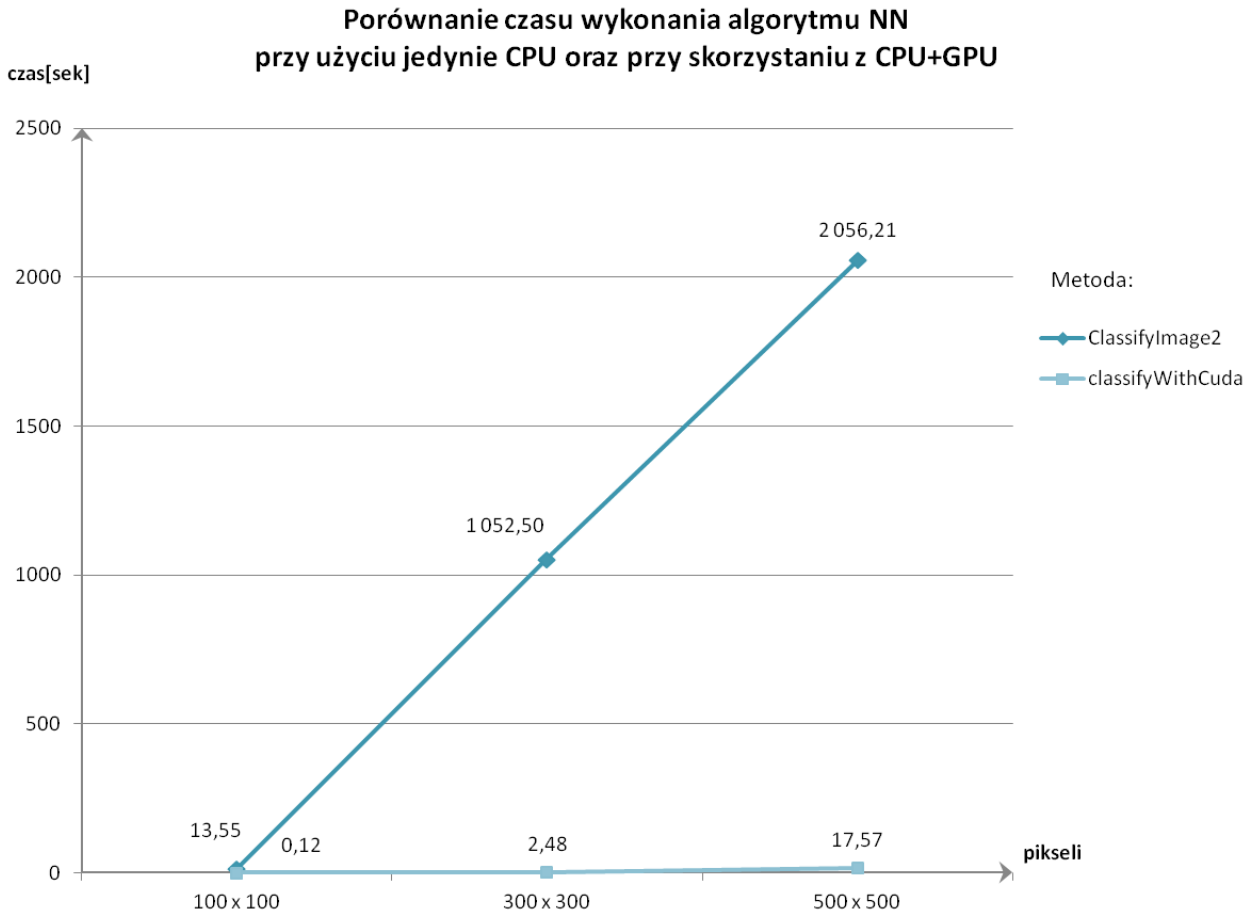
Sprawdzana metoda	Obraz wzorcowy	Kappa	I_{CP}
classifyWithCuda	*ClassifyImage2*	1	1
classifyWithCudakNN	*ClassifyImage2kNN*	1	1
classifyWithCudaTex	*ClassifyImage2*	0,3014	0,4490
classifyWithCudakNNWithGroup	*ClassifyImage2kNN*	1	1

Tabela 2 Wyniki walidacji klasyfikacji wykonanej metodami równoległymi
(źródło: opracowanie własne)

9. Konfrontacja podejścia sekwencyjnego z równoległym

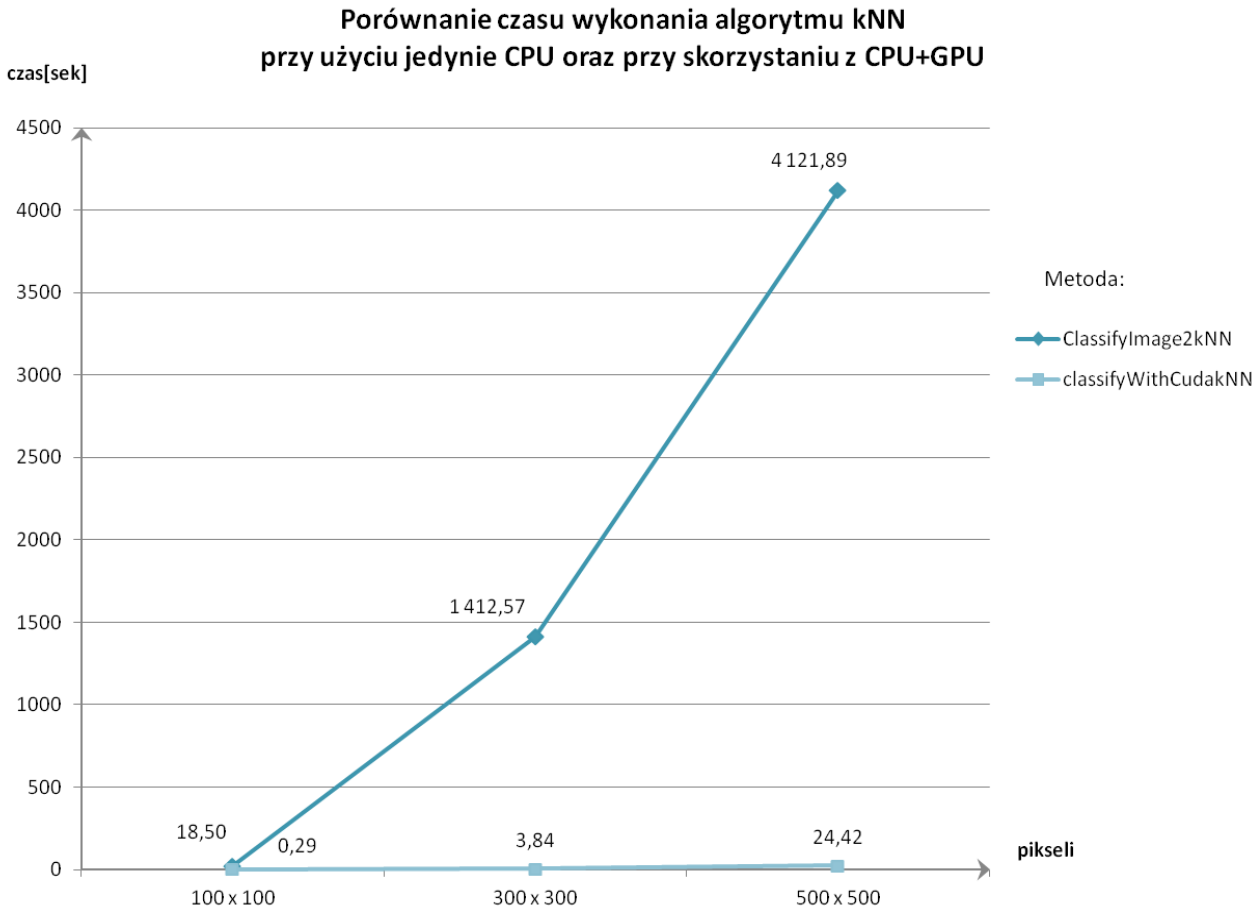
Głównym celem pracy było zbadanie możliwości optymalizacji klasyfikacji nadzorowanej obrazów teledetekcyjnych przy zastosowaniu programowania równoległego na procesorach graficznych. Do porównywania wydajności algorytmów sekwencyjnych i paralelnych użyto najkorzystniejszych czasowo, poprawnych, uzyskanych w trakcie badań rozwiązań, odpowiednio dla CPU oraz GPU.

Dla algorytmu najbliższego sąsiada najbardziej optymalną metodą na CPU okazała się *ClassifyImage2*. Czas jej wykonania wzrasta proporcjonalnie do wielkości danych przeznaczonych do przetworzenia. To samo zjawisko ma miejsce w przypadku najlepszej uzyskanej metody w podejściu heterogenicznym *classifyWithCuda*. Przyrost czasu niezbędnego do wykonania obliczeń jest jednak radykalnie inny. Stosując w obliczeniach procesor graficzny, uzyskano od kilku do ponad sto razy lepsze rezultaty niż używając jedynie procesora CPU(*Wykres 7*).



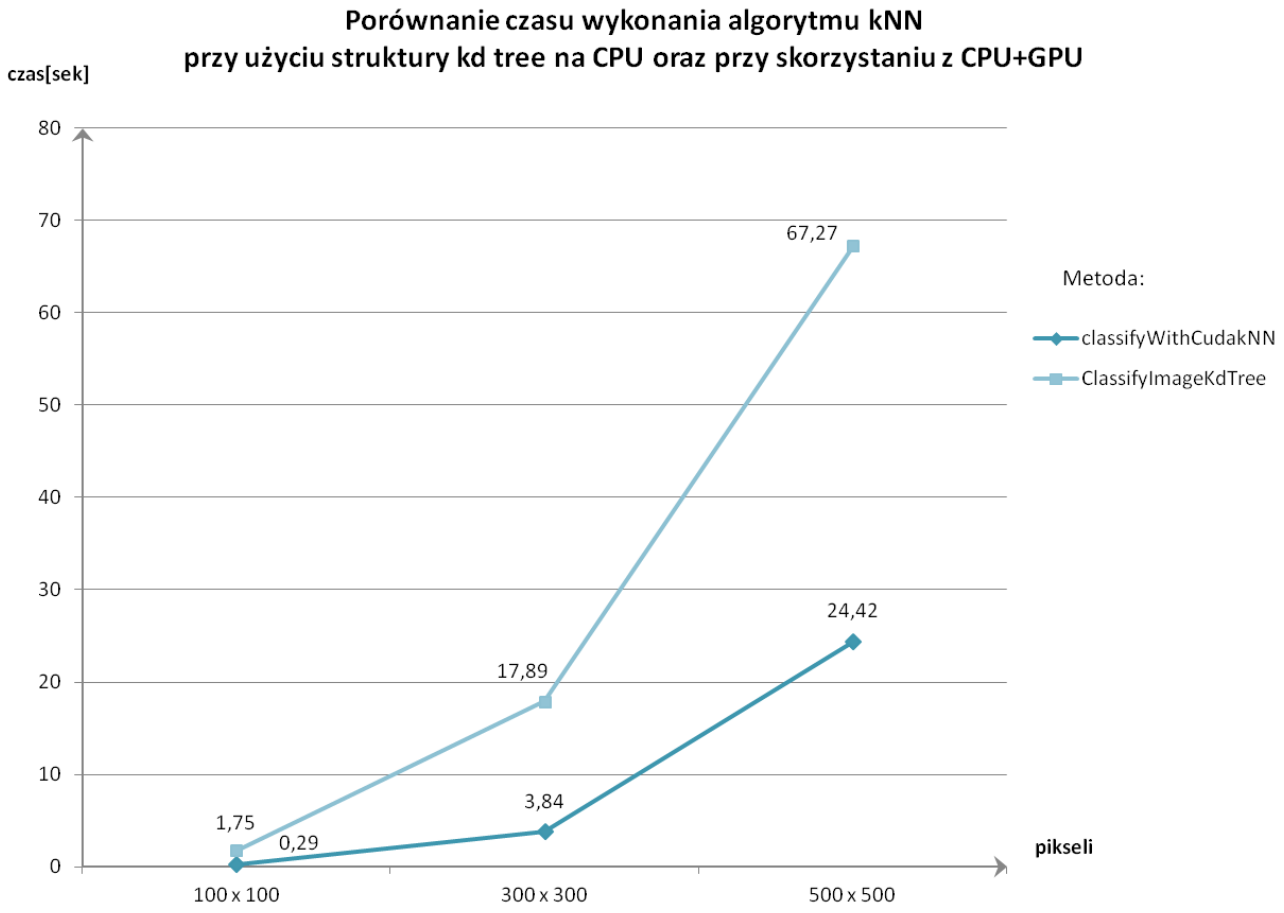
Wykres 7 Porównanie podejścia sekwencyjnego i równoległego w metodzie NN (źródło: opracowanie własne)

Podobne zależności otrzymano w algorytmie k-najblizszych sąsiadów, porównując metodę ClassifyImage2kNN oraz classifyWithCudakNN. Również tutaj równoległe wykonywanie obliczeń okazało się zdecydowanie lepszym rozwiązaniem(*Wykres 8*). Należy zauważyć, że im więcej danych posiadamy do sklasyfikowania tym większy uzyskamy zysk względem podejścia sekwencyjnego.



Wykres 8 Porównanie podejścia sekwencyjnego i równoległego w metodzie kNN (źródło: opracowanie własne)

Kolejne pytanie, na jakie powinniśmy sobie odpowiedzieć, to czy programowanie na GPU jest w stanie przewyższyć wydajnością rozwiązania, które są powszechnie stosowane do akceleracji obliczeń związanych z klasyfikacją obrazów teledetekcyjnych. Z przeprowadzonych badań wynika jednoznacznie, że metoda ClassifyWithCudakNN dała lepsze rezultaty niż metoda ClassifyImageKdTree. Oznacza to, że bardziej korzystne jest zrównoleglenie algorytmu kNN przy użyciu procesorów graficznych niż skorzystanie ze struktury k-wymiarowego drzewa.

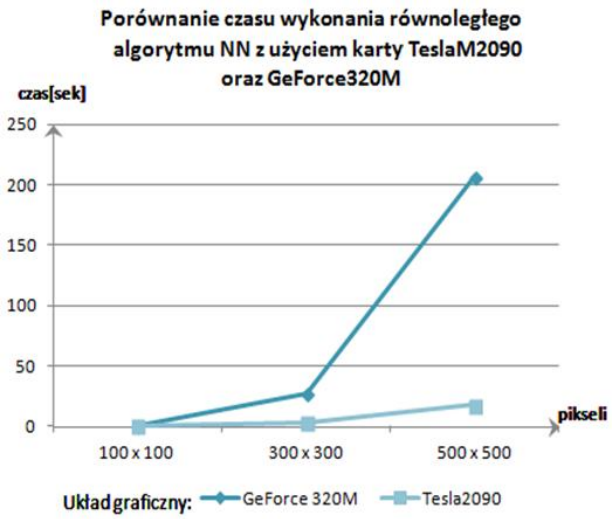


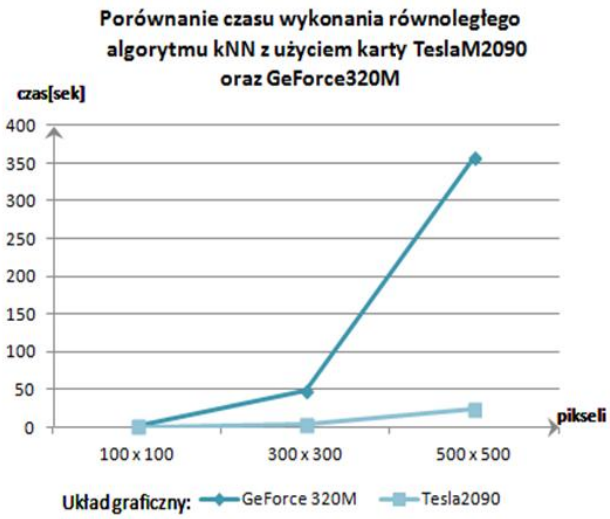
Wykres 9 Porównanie akceleracji obliczeń za pomocą procesora graficznego oraz z użyciem kd drzewa (źródło: opracowanie własne)

10. Porównanie wyników dla GeForce GT 320M i TeslaM2090

Badanie wydajności obliczeń równoległych przeprowadzono dodatkowo na kolejnym układzie graficznym – GeForce GT320M. Zgodnie z *Tabelą 1*. Jest to urządzenie odznaczające się nieco słabszymi parametrami niż, użyta do poprzednich akceleracji, TeslaM2090. Układ GeForce posiada znacznie mniejszą liczbę multiprocesorów. Jest ich aż o 488 mniej niż w przypadku Tesli M2090. W momencie, gdy na mocniejszej karcie graficznej możliwe jest swobodne zrównoleglenie obliczeń o określonej liczbie wątków, na słabszej może zajść konieczność wykonania tego zadania partiami. Jeśli procesor nie jest w stanie wykonać wszystkich wątków równolegle, wykonuje ich jednocześnie tylko tyle, na ile pozwala mu ilość multiprocesorów. Reszta oczekuje na swoją kolej. Ponadto układ GeForce odznacza się mniejszą przepustowością magistrali pamięci, co powoduje wolniejsze dostarczanie i odbieranie danych przez procesor graficzny. Wpływ na czas obliczeń miała też

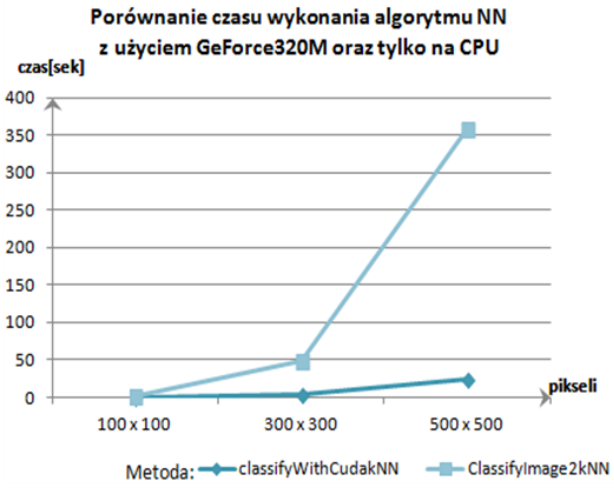
z pewnością wielkość poszczególnych obszarów pamięci układu. W związku z powyższymi zależnościami, układ GeForce uzyskał mniejszą wydajność niż Tesla (*Wykresy 10 i 11*).

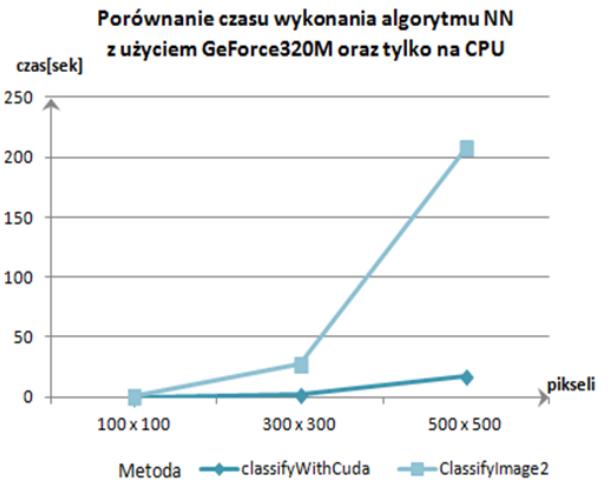




Wykresy 10,11 Porównanie wydajności karty Tesla M2090 oraz GeForce GT 320M
(źródło: opracowanie własne)

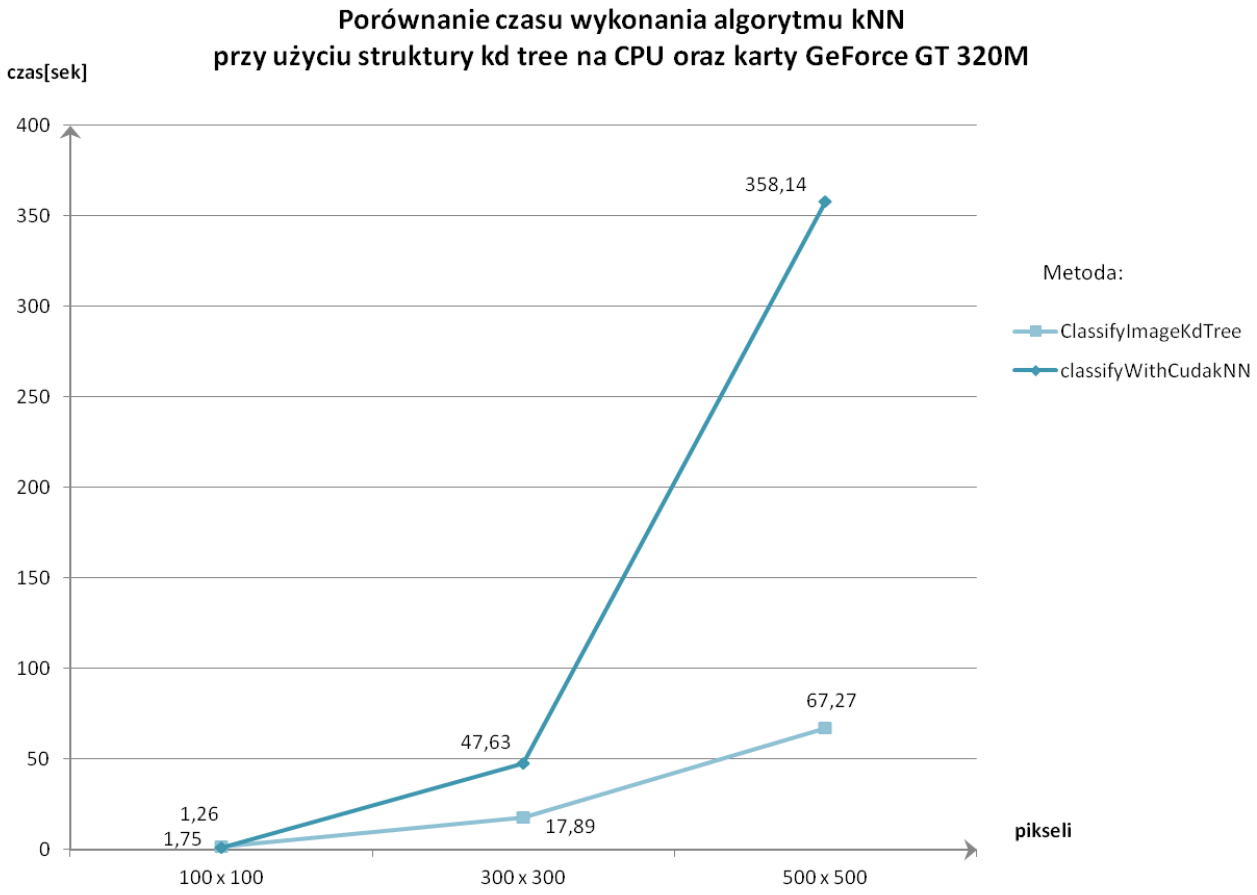
Wydajność metod, uruchomionych na procesorze karty GeForce GT 320M, porównano również z odpowiadającymi metodami wykonanymi na CPU. Rozwiązania heterogeniczne, wykonane nawet przy użyciu słabszych kart graficznych, wciąż mają dużą przewagę czasową nad obliczeniami sekwencyjnymi(*Wykresy 12,13*).





Wykresy 11,12 Porównanie podejścia sekwencyjnego i równoległego dla karty GeForce GT 320M
(źródło: opracowanie własne)

Jednakże, procesor GeForce GT 320M, jako urządzenie starszej generacji, o słabszych parametrach niż Tesla 2090M, nie sprawdziło się we wszystkich realizowanych testach. Akceleracja obliczeń przy użyciu tego układu graficznego okazała się być mniej korzystnym rozwiązaniem niż zastosowanie struktury k-wymiarowego drzewa(*Wykres 14*).



Wykres 12 Porównanie akceleracji obliczeń za pomocą GPU GeForce GT 320M oraz z użyciem kd drzewa (źródło: opracowanie własne)

11. Podsumowanie i wnioski

Niniejsza praca magisterska miała na celu zbadanie możliwości optymalizacji klasyfikacji nadzorowanej obrazów teledetekcyjnych przy zastosowaniu procesorów graficznych i programowania równoległego. Przeprowadzono w związku z tym szereg prób implementacyjnych, prowadzących do uzyskania jak najlepszego czasu wykonania przedmiotowego zadania. W badaniu skupiono się na algorytmach najbliższego sąsiada i k-najbliższych sąsiadów. Każdy z nich został przygotowany w dwóch wersjach – sekwencyjnej na CPU oraz heterogenicznej. Przy czym algorytm kNN,

wykonywany w klasyczny sposób, został dodatkowo akcelerowany poprzez skorzystanie ze struktury kd drzewa. Z przeprowadzonych analiz można wyciągnąć ogólny wniosek, że ze względu na swój wielowątkowy charakter obliczeń, procesor GPU wykazał się znacznie większa wydajnością niż CPU w opracowywanych algorytmach.

Badania pokazały, że pomimo iż procesor graficzny posiada duży potencjał obliczeniowy, to wykorzystanie go w prawidłowy sposób nie jest prostym zadaniem. Obliczenia, realizowane przy użyciu architektury SIMT, są znacznie trudniejsze w implementacji niż obliczenia sekwencyjne. W przypadku korzystania z procesorów graficznych, programista musi mieć na uwadze specyficzny sposób organizacji pracy wątków oraz zarządzania pamięcią. Często wąskim gardłem aplikacji, wprowadzającym pewne ograniczenia w organizacji kodu, bywa też przepustowość magistrali przesyłu danych między CPU a GPU. Należy więc uprzednio poznać i zrozumieć istotę technik programowania heterogenicznego.

Podczas eksploracji technologii równoległych zauważono, iż czynnikiem decydującym o stopniu akceleracji jest w dużej mierze ilość wątków w obrębie jednego bloku. Istnieje generalna zależność, że im ich więcej, tym obliczenia wykonają się szybciej. Jednakże zawsze najlepszym sposobem jest empiryczne sprawdzenie jaka ich ilość jest najkorzystniejsza. Wśród implementacji, dedykowanych procesorowi graficznemu, jedynie metoda wykorzystująca pamięć tekstur okazała się niepomyślna. Użytkowanie tego obszaru pamięci nie przyniosło zakładanego wzrostu prędkości, a dodatkowo został zwrócony zły wynik klasyfikacji. Pozostałe próby zakończyły się pełnym sukcesem. Rezultat ich działania zgadzał się całkowicie z wynikami obliczeń sekwencyjnych, a czas wykonania był o wiele krótszy. Obliczenia heterogeniczne, odpowiadające bezpośrednio sekwencyjnym metodą, wykonały się od nich wielokrotnie szybciej, a przyrost przyspieszenie rośnie wraz z ilością danych.. Porównując natomiast pracje z GPU do metody korzystającej z k-wymiarowego drzewa, można było zaobserwować jak duże znaczenie ma sam sprzęt, jakim dysponujemy. W przypadku karty GeForce GT 320M uzyskano dłuższy czas przetwarzania danych. Natomiast użycie karty Tesla2090 spowodowało już znaczne przyspieszenie w obliczeniach w porównaniu z metodą wykorzystującą strukturę drzewa. Ostatni wynik świadczy o tym, że jeśli posiadamy dobrej klasy GPU, to bardziej opłaca się zrównoleglić algorytm klasyfikacji niż skorzystać ze złożonych struktur przechowywania danych.

Należy zaznaczyć, iż klasyfikacja nadzorowana jest procesem, który da się rozdzielić pomiędzy pracę wielu multiprocesorów. Dzięki temu możliwe było skorzystanie z dobrodziejstw karty graficznej. Zadania iteracyjne nie wykorzystują już bowiem mocy obliczeniowej GPU, a czas przesyłu danych wydłuża tylko cały proces. Zauważyć to można było w przypadku metody kNN, gdzie podczas próby przeprowadzenia części sekwencyjnej obliczeń na GPU, czas wykonania zadania znacznie się wydłużył.

Przeprowadzone badania i uzyskane w nich wyniki potwierdzają postawioną we wstępie tezę, iż równoległe wykonanie zadania klasyfikacji nadzorowanej na GPU przyniesie znaczny wzrost wydajności w stosunku do rozwiązania klasycznego na CPU. W pracy dowiedziono również, że jeśli dysponuje się odpowiednio zaawansowanymi urządzeniami, użycie procesora graficznego jest lepszym rozwiązaniem niż stosowanie dotychczas szeroko stosowanej struktury k-wymiarowego drzewa. Obliczenia heterogeniczne z udziałem układów GPU stają się stopniowo takim samym standardem jak procesory wielordzeniowe. Widać to w szczególności we współcześnie prowadzonych badaniach naukowych. Należy rozwijać ten sposób przetwarzania danych w zadaniach geoinformatycznych.

12. Bibliografia

1) Szymczyk M., Szymczyk P., 2011. Możliwość zastosowania GPU do przetwarzania obrazów dla celów analizy sceny, AGH, Automatyka Tom 15 Zeszyt 3, Kraków
2) Jabłoński M., Bubliński Z., 2009 Akceleracja algorytmów przetwarzania obrazów z wykorzystaniem zasobów karty graficznej, AGH, Automatyka Tom 13 Zeszyt 3, Kraków
3) Sanders J., Kandrot E., 2012 CUDA w przykładach, Helion, Gliwice
4) Bernstein A.J., 1966. Program Analysis for Parallel Processing, IEEE Trans. On Electronic Computers, EC-15, s. 757–762
5) Amdahl G.M. , 1967. Validity of single-processor approach to achieving large-scale computing capability, Proceedings of AFIPS Conference, Reston, s. 483-485
6) Gustafson J.L., 1988. Reevaluating Amdahl's Law, CACM, s. 532-533
7) NVIDIA Corporation, 2015. NVIDIA CUDA C Programming Guide, wersja 4.2 , http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, Santa Clara
8) NVIDIA Corporation, 2015. NVIDIA NVIDIA CUDA C Best Practices, wersja 4.2 , http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, Santa Clara
9) Mokrzycki W., 1992. Encyklopedia przekształcania obrazów, Akademicka Oficyna Wydawnicza RM, Warszawa
10) Zagajewski B., Jarocińska A, Olesiuk D., 2010. Metody i techniki badań geoinformatycznych, Uniwersytet Warszawski, Warszawa
11) Sitek Z., 1992. Zarys teledetekcji lotniczej i satelitarnej, AGH, Skrypt 1239, Kraków
12) Adamczyk J., Bedkowski K., 2007. Metody cyfrowe w teledetekcji, Szkoła Główna Gospodarstwa Wiejskiego, Warszawa
13) Berg M. , Kreveld M., Overmars M., 2008. Computational geometry: algorithms and applications, Otfried Cheong
14) Landsat Project Description, 2015. http://landsat.usgs.gov
15) Managed CUDA documentation, 2015. , https://managedcuda.codeplex.com
16) Sintrom E., Assarsson U., 2008, Fast parallel gpu-sorting a hybrid algorithm, Chalmers University Of Technology Gothenburg, Sweden
17) Kuang Q., Zhao L., 2009. A Practical GPU Based KNN Algorithm, Soochow Uniwersity, Huangshan

13. Spis rysunków

14. Spis tabel

15. Spis wykresów

16. Spis listingów

ZAŁĄCZNIK 1

Nazwa metody	Wątki na Blok	Rozdzielczość obrazu	GPU	CPU	CZAS
ClassifyImage	1024	100	TeslaM2090	Intel E5651	00:01:03.0285673
ClassifyImage2	1024	100	TeslaM2090	Intel E5651	00:00:13.5549955
ClassifyImage2kNN	1024	100	TeslaM2090	Intel E5649	00:00:18.4987439
ClassifyImageKdTree	1024	100	TeslaM2090	Intel E5650	00:00:01.7547884
classifyWithCuda	1024	100	TeslaM2090	Intel E5651	00:00:00.1205749
classifyWithCudakNN	1024	100	TeslaM2090	Intel E5652	00:00:00.2882515
classifyWithCudaTex	1024	100	TeslaM2090	Intel E5653	00:00:00.1614585
classifyWithCudakNNWithGroup	1024	100	TeslaM2090	Intel E5654	00:00:00.6410577
ClassifyImage	1024	300	TeslaM2090	Intel E5656	01:20:06.8189868
ClassifyImage2	1024	300	TeslaM2090	Intel E5657	00:17:32.5029488
ClassifyImage2kNN	1024	300	TeslaM2090	Intel E5658	00:23:32.5679766
ClassifyImageKdTree	1024	300	TeslaM2090	Intel E5659	00:00:17.8929567
classifyWithCuda	1024	300	TeslaM2090	Intel E5660	00:00:02.4781400
classifyWithCudakNN	1024	300	TeslaM2090	Intel E5661	00:00:03.8415050
classifyWithCudaTex	1024	300	TeslaM2090	Intel E5662	00:00:03.2391421
classifyWithCudakNNWithGroup	1024	300	TeslaM2090	Intel E5663	00:01:09.1534748
ClassifyImage2	1024	500	TeslaM2090	Intel E5666	02:22:16.2092470
ClassifyImage2kNN	1024	500	TeslaM2090	Intel E5667	02:56:41.8852839
ClassifyImageKdTree	1024	500	TeslaM2090	Intel E5668	00:01:07.2663227
classifyWithCuda	1024	500	TeslaM2090	Intel E5669	00:00:17.5728844
classifyWithCudakNN	1024	500	TeslaM2090	Intel E5670	00:00:24.4172414
classifyWithCudaTex	1024	500	TeslaM2090	Intel E5671	00:00:22.7035230
classifyWithCudakNNWithGroup	1024	500	TeslaM2090	Intel E5672	00:13:18.4141093
classifyWithCuda	1024	300	TeslaM2090	Intel E5674	00:00:02.4781400
classifyWithCuda	1024	500	TeslaM2090	Intel E5675	00:00:17.5728844
classifyWithCuda	512	300	TeslaM2090	Intel E5676	00:00:02.3228992
classifyWithCuda	512	500	TeslaM2090	Intel E5677	00:00:17.0770696
classifyWithCuda	256	300	TeslaM2090	Intel E5678	00:00:02.2843688

classifyWithCuda	256	500	TeslaM2090	Intel E5679	00:00:17.0149479
classifyWithCuda	128	300	TeslaM2090	Intel E5680	00:00:02.2891080
classifyWithCuda	128	500	TeslaM2090	Intel E5681	00:00:16.8064361
classifyWithCuda	64	300	TeslaM2090	Intel E5682	00:00:02.4696879
classifyWithCuda	64	500	TeslaM2090	Intel E5683	00:00:18.3670237
classifyWithCuda	32	300	TeslaM2090	Intel E5684	00:00:03.6936707
classifyWithCuda	32	500	TeslaM2090	Intel E5685	00:00:27.4688148
classifyWithCuda	16	300	TeslaM2090	Intel E5686	00:00:06.7975923
classifyWithCuda	16	500	TeslaM2090	Intel E5687	00:00:52.0954080
classifyWithCuda	8	300	TeslaM2090	Intel E5688	00:00:13.0139201
classifyWithCuda	8	500	TeslaM2090	Intel E5689	00:01:40.6131217
classifyWithCuda	4	300	TeslaM2090	Intel E5690	00:00:24.8672082
classifyWithCuda	4	500	TeslaM2090	Intel E5691	00:03:14.7258932
classifyWithCuda	512	100	GeForce	Intel Core i3	00:00:00.4529770
classifyWithCudakNN	512	100	GeForce	Intel Core i4	00:00:01.2643533
classifyWithCuda	512	300	GeForce	Intel Core i5	00:00:27.1392606
classifyWithCudakNN	512	300	GeForce	Intel Core i6	00:00:47.6330397
classifyWithCuda	512	500	GeForce	Intel Core i7	00:03:27.6896125
classifyWithCudakNN	512	500	GeForce	Intel Core i8	00:05:58.1364509