



POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRONIKI I TECHNIK INFORMACYJNYCH
INSTYTUT INFORMATYKI

Rok akademicki
2011/2012

PRACA DYPLOMOWA MAGISTERSKA

DAMIAN TURCZYŃSKI

OPTYMALIZACJA INTERFEJSU KOMPUTER-UŻYTKOWNIK Z UŻYCIEM PROCESORA
GRAFICZNEGO

Opiekun pracy:
mgr inż. Krzysztof Gracki

Ocena:

.....
Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego



DAMIAN TURCZYŃSKI

Specjalność: Inżynieria Systemów Informatycznych

Data urodzenia: 09.10.1987 r.

Data rozpoczęcia studiów: 01.10.2010 r.

ŻYCIORYS

Urodziłem się 09.10.1987 r. w Mielcu. Po ukończeniu szkoły podstawowej i gimnazjum naukę kontynuowałem w IX Liceum Ogólnokształcącym im. Bohaterów Monte Cassino w Szczecinie. W szkole średniej uczęszczałem do klasy o profilu informatyczno-matematycznym. W latach 2006 - 2010 zdobyłem tytuł inżyniera na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. W roku 2010 rozpoczęłem studia magisterskie na tym samym wydziale.

.....
Damian Turczyński

EGZAMIN DYPLOMOWY

Złożył egzamin dyplomowy w dniu 2012 r

z wynikiem

Ogólny wynik studiów:

Dodatkowe wnioski i uwagi Komisji:

.....

.....

Optymalizacja interfejsu komputer-użytkownik z użyciem procesora graficznego

Streszczenie

Praca traktuje o optymalizacji oraz usprawnieniu działania interfejsu użytkownik-komputer. Optymalizowany został program służący do rozpoznawania gestów i ruchu dloni za pomocą kamery internetowej stworzony przez autora w ramach pracy inżynierskiej. Praca zawiera wstęp teoretyczny, opis rozwiązania oraz podsumowanie osiągniętych wyników.

Praktyczną część stanowi implementacja systemu uwzględniająca działanie aplikacji z użyciem procesora graficznego. W niniejszym dokumencie przedstawiono, postawione wymagania i cel tworzonego systemu, szczegółowo opisane rozwiązanie oraz wyniki przeprowadzonych eksperymentów wraz z wnioskami.

Na końcu dokumentu znajdują się dodatki uwzględniające zrzuty ekranu aplikacji, kod źródłowy wybranych algorytmów oraz omówienie biblioteki OpenCV oraz rozwiązania NVIDIA® CUDA™.

Słowa kluczowe: *interfejs, rozpoznawanie, gpu, przetwarzanie równoległe*

Computer - user interface optimization using GPU

Abstract

This thesis refers to the optimization and improvement of user-computer interface. Optimized system is used to recognize hand gestures and movements using a webcam and it was created by the author during engineering dissertation. This document is a descriptive part.

The practical part is the implementation of the system that is using the GPU and include upgrades. This paper consist of a theoretical introduction, requirements of the created system, details of the solution and the results of the study and the conclusions.

At the end of the document there are appendixes that include application screenshots, source code of selected algorithms and description of OpenCV library and the NVIDIA® CUDA™ solution.

Keywords: *hand, interface, recognition, gpu, parallel computing*

Spis treści

| | |
|---|-----------|
| 1 Wstęp | 6 |
| 2 Cel i zakres pracy | 8 |
| 2.1 Zakres pracy inżynierskiej | 8 |
| 2.2 Cele nowego systemu | 9 |
| 2.3 Przegląd pracy | 10 |
| 3 Podstawy teoretyczne, przegląd literatury | 12 |
| 3.1 Opis metod zrównoleglania kodu | 12 |
| 3.1.1 Rozwój obliczeń równoległych na podstawie przetwarzania obrazów | 13 |
| 3.1.2 Podstawowe problemy przetwarzania równoległego | 15 |
| 3.1.3 Modele spójności | 16 |
| 3.1.4 Prawo Amdahla | 17 |
| 3.1.5 Rodzaje zrównoleglania kodu | 18 |
| 3.1.6 Taksonomia Flynn'a | 19 |
| 3.1.7 Programowanie równoległe na procesorach wielordzeniowych | 20 |
| 3.1.8 Przykładowe programy czerpiące korzyść ze zrównoleglenia kodu | 22 |
| 3.1.9 Podsumowanie | 23 |
| 3.2 Przetwarzanie obrazów | 24 |
| 3.2.1 Podstawowe algorytmy przetwarzania obrazów | 24 |
| 3.2.2 Algorytmy segmentacji tła | 27 |
| 3.3 Interfejsy użytkownika z wykorzystaniem sieci komputerowych | 29 |
| 3.3.1 Komunikacja pomiędzy modułami systemu | 29 |
| 3.3.2 Architektura klient serwer | 30 |
| 3.3.3 Programowanie gniazd | 31 |
| 3.3.4 Istniejące przykłady rozwiązań | 32 |
| 3.3.5 Podsumowanie | 33 |

| | |
|---|-----------|
| 4 Proponowane rozwiązanie systemu | 34 |
| 4.1 Wymagania i założenia | 34 |
| 4.2 Algorytmy przetwarzania równoległego | 37 |
| 4.2.1 Autorski algorytm detekcji skóry | 38 |
| 4.2.2 Zróżnicowane algorytmy morfologii | 45 |
| 4.2.3 Zróżnicowanie programu jako całości | 51 |
| 4.3 Algorytm wydzielania statycznego tła | 59 |
| 4.3.1 Wymagania i cel | 60 |
| 4.3.2 Zastosowane rozwiązanie | 60 |
| 4.3.3 Zróżnicowanie wydzielania tła | 61 |
| 4.3.4 Badania rozwiązania i wyniki | 62 |
| 4.3.5 Wnioski | 62 |
| 4.4 Algorytm wydzielania dloni z całej ręki | 64 |
| 4.4.1 Wymagania i cel | 64 |
| 4.4.2 Zastosowane rozwiązanie | 64 |
| 4.5 Sieciowy interfejs programu | 65 |
| 4.5.1 Opis zastosowanego rozwiązania | 66 |
| 4.5.2 Przykładowy klient | 67 |
| 5 Podsumowanie | 68 |
| Bibliografia | 71 |
| Dodatek A Zrzuty ekranu działającej aplikacji | 73 |
| Dodatek B Kod źródłowy rozwiązań autorskich | 76 |
| B.1 Pierwszy krok zdekomponowanej erozji binarnej na procesorze graficznym. | 76 |
| B.2 Drugi krok zdekomponowanej erozji binarnej na procesorze graficznym. | 77 |
| B.3 Filtr wstępny segmentacji koloru skóry. | 77 |
| B.4 Jedna z funkcji segmentacji skóry. | 78 |
| B.5 Algorytm segmentacji statycznego tła z obrazu. | 79 |
| B.6 Algorytm wydzielenia dloni z obszaru. | 81 |
| Dodatek C OpenCV oraz NVIDIA® CUDA™ Przegląd i zastosowania praktyczne | 83 |
| C.1 Wstęp | 83 |
| C.2 OpenCV | 84 |
| C.2.1 Główne cechy | 84 |

| | | |
|-------|--------------------------------------|----|
| C.2.2 | Moduły | 84 |
| C.2.3 | Zastosowanie | 85 |
| C.2.4 | Przykładowy kod | 85 |
| C.3 | NVIDIA® CUDA™ | 86 |
| C.3.1 | Główne cechy | 86 |
| C.3.2 | Technologia | 87 |
| C.3.3 | Interfejsy programistyczne | 87 |
| C.4 | CUDA™ wraz z OpenCV | 88 |
| C.4.1 | Kod źródłowy | 88 |
| C.5 | Inne rozwiązania | 90 |
| C.6 | Podsumowanie | 90 |
| C.7 | Bibliografia | 92 |

Rozdział 1

Wstęp

Obecnie obserwuje się tworzenie nowych interfejsów użytkownika do komunikacji zarówno z komputerem jak również innymi urządzeniami elektronicznymi. Klawiatura i mysz komputerowa w dalszym ciągu są jednak najczęściej używane. Mimo to w ostatnich latach, głównie za sprawą dotykowych telefonów komórkowych, rozpowszechnił się nowy interfejs - dotykowy. Człowiek intuicyjnie nie chce używać dodatkowych urządzeń ponad to, czym wyposażał go natura. W tym przypadku to jego dlonie są narzędziem do przekazywania informacji. Co jednak, gdy chcielibyśmy pójść o krok dalej i nie musieć nawet niczego dotykać?

W tym miejscu należy zaprezentować nowy, zyskujący ostatnio na popularności, sposób na komunikację ze światem elektroniki. Są nim gesty i ruch dloni w przestrzeni. Jeżeli dołączy się do tego rzeczywiste wyświetlanie w trój wymiarze, na przykład holograficzne, można już dziś sformułować hipotetyczną wizję jutra. To czy właśnie ten sposób przypadnie do gustu ludziom na całym świecie pokaże czas.

Już dziś jednak widać, że w dziale rozrywki używanie ciała człowieka, jako kontrolera przyjęło się znakomicie. Świadczy o tym popularność Kinect'a firmy Microsoft®. Warto jednak zaznaczyć, że rozpowszechnienie Kinect'a pozwoliło na powstanie rozwiązań również w innych dziedzinach jak na przykład w medycynie. W magazynie New Scientist z 19 maja 2012 [20] roku przedstawiono użycie Kinect'a do rozpoznawania pozycji ciała lekarza w celu sterowania przez niego maszynami wykonującymi rentgen bądź tomograf komputerowy.

W przypadku tworzenia systemów wykorzystujących ten typ interfejsów należy rozwiązać szereg trudności wynikających z jego charakteru. Łatwo jest przetworzyć dokładny ruch myszy komputerowej na impuls elektryczny jak również łatwo odebrać informację z klawiatury. W przypadku interfejsów opartych na detekcji ruchu ludzi nie ma możliwości podłączenia sensorów do człowieka by nie utrudniało mu to poruszania się.

Przecież nikt z nas nie chciałby chodzić z podłączonymi do ręki kablami, bądź sensorami. Dlatego wymagania dla takich systemów wynikają głównie z użyteczności systemu.

System spełniający zadanie interfejsu użytkownika opartego o ciało ludzkie musi spełniać dwa podstawowe warunki:

- Rozpoznawanie gestów, ruchu musi być bardzo dokładne i niezawodne.
- Responsywność systemu musi być na bardzo wysokim poziomie.

Programiści starają się rozwiązać oba powyższe problemy na różne sposoby. Jednym z nich jest tworzenie wyspecjalizowanych urządzeń (na przykład Kinect) do badania pozycji ciała człowieka, a następnie odbiór i przetwarzanie tych dokładniejszych danych w programach. Drugim sposobem jest użycie podstawowych urządzeń takich jak zwykła kamera internetowa, bardzo powszechna i łatwo dostępna, oraz napisanie wyspecjalizowanego oprogramowania do przetwarzania obrazów i rozpoznawania ruchu oraz gestów.

W pracy skupiono się na drugim podejściu w celu zwiększenia potencjalnych odbiorców. Starano się uzyskać jak najlepsze wyniki rozpoznawania przy jednoczesnym zachowaniu wysokiej wydajności działania programu. W pracy rozwijano rozwiązanie z pracy dyplomowej inżynierskiej w celu optymalizacji wydajności i jakości rozpoznawania.

Omówione zostały sposoby przetwarzania równoległego, oraz zaimplementowany został system, w głównej mierze działający na procesorze graficznym. Dzięki takiemu rozwiązaniu odciąża się procesor główny, a co za tym idzie, system ma mniejsze wymagania wydajnościowe. W pracy przebadano możliwość zrównoleglenia każdego kroku algorytmu, oraz zaimplementowano wszystkie części, których wykonanie na procesorze graficznym przynosiło korzyść.

Dodatkowo skupiono się na poprawie jakości rozpoznawania gestów poprzez wyeliminowanie problemów i niedoskonałości z poprzedniej wersji systemu. Głównym zagadnieniem wymagającym uwagi był algorytm wydzielania statycznego tła. Ponadto usprawniono interfejs komunikacji z innymi programami oraz dodano szereg mniejszych usprawnień.

Rozdział 2

Cel i zakres pracy

Głównym celem pracy jest stworzenie optymalnego programu służącego, jako interfejs użytkownika. Do uzyskania tego założenia należy przeprowadzić analizę istniejących rozwiązań oraz zaproponować i zaimplementować aplikację, która odznaczać się będzie lepszymi wynikami aniżeli ta stworzona w trakcie pracy inżynierskiej.

Praca bazuje na wynikach osiągniętych przez autora w ramach pracy inżynierskiej. Tematem tej pracy było stworzenie programu służącego do komunikacji człowieka z komputerem przy użyciu kamery internetowej w czasie rzeczywistym [22].

2.1 Zakres pracy inżynierskiej

W ramach pracy inżynierskiej przebadano i rozwiązano następujące problemy:

- Metoda segmentacji dloni na podstawie modelu barwy ludzkiej skóry.
- Opracowanie metody analizy gestów wykonywanych dlonią.
- Metoda wykrywania i śledzenia ruchu dloni.
- Wysyłanie komunikatów do systemu przekladających gesty i ruch dloni na zdarzenia imitujące ruch i kliknięcia myszy komputerowej.

W pracy posługiwano się kolorowym obrazem pobieranym z kamery o rozdzielcości 320 na 240 pikseli. Z obrazu w pierwszym etapie zostawały wyodrębnione piksele mogące należeć do skóry, za pomocą autorskich algorytmów działających w różnych przestrzeniach barw. Algorytmy te były główną częścią pracy inżynierskiej i zostały dokładnie przebadane. W kolejnym etapie rozpoznawano gest dloni przy użyciu klasyfikatora Bayesa, maszyny wektorów nośnych oraz klasyfikatora najbliższych sąsiadów. Kolejnym etapem było wykrycie i śledzenie ruchu dloni za pomocą autorskiego algorytmu. Przygotowane



Rysunek 2.1. Gest neutralny.



Rysunek 2.3. Gest A.



Rysunek 2.2. Gest R.



Rysunek 2.4. Gest T.

w pracy zbiory uczące pozwalają na wykrywanie czterech gestów. Są to gest A, R oraz T z polskiego alfabetu palcowego oraz otwarta dłoń jako gest neutralny (rysunki 2.1 - 2.4). Wyniki osiągane przez program były przekazywane do systemu operacyjnego jako komunikaty ruchu i kliknięć myszy komputerowej. Dzięki takiemu rozwiązaniu możliwa była obsługa komputera jedynie za pomocą gestów i ruchu dłoni przed kamerą internetową.

W pracy inżynierskiej zastosowano prosty i dający słabe wyniki algorytm wydzielania statycznego tła. Oznaczał się on dużą podatnością na zakłócenia. Dodatkowo istniało wymaganie by użytkownik systemu nosił ubranie z długim rękawem w celu umożliwienia detekcji obszaru dłoni przez system.

2.2 Cele nowego systemu

Cele nowego systemu można podzielić na dwie grupy. Pierwsza z nich, to grupa obejmująca algorytmy odpowiedzialne za poprawienie jakości rozpoznawania, czyli jak najlepsze spełnienie warunku pierwszego w poprzednim rozdziale. Druga grupa postawionych celów obejmuje szeroko zakrojoną optymalizację systemu. W drugim punkcie zawiera się zarówno optymalizacja pojedynczych metod jak i sposobu przetwarzania aplikacji jako całości.

Do pierwszej grupy celów należy wyeliminowanie niedogodności z pracy inżynierskiej polegającej na wymuszeniu używania ubrania z długim rękawem w celu poprawnego

działania aplikacji. Dodatkowo poprawieniu musi ulec algorytm segmentacji tła z obrazu, jako że zaimplementowany algorytm posiada wiele wad i niedoskonałości.

Dodatkowo przebadane i dopasowane muszą zostać wszystkie parametry w aplikacji w celu ustalenia zmiennych sterujących działaniem poszczególnych algorytmów do zastosowań rzeczywistych i uwzględniających wszelkie możliwe scenariusze działań.

Główną częścią pracy jest spełnienie drugiej grupy celów, czyli optymalizacja systemu. Zdecydowano się, że dla poprawy wydajności i zmniejszenia zajętości zasobów systemowych przez aplikację przetwarzanie programu, w głównej mierze, zostanie przeniesione na kartę graficzną posiadającą dużo więcej wątków. Dzięki temu można mówić o zrównolegleniu aplikacji i ten termin będzie używany w pracy wielokrotnie, a będzie dotyczył właśnie przeniesienia obliczeń na procesor graficzny.

Oprócz części implementacyjnej, bardzo ważna jest część dotycząca badań. Postawione w poprzednim paragrafie cele prowadzą do tezy, że przetwarzanie systemu służącego jako interfejs użytkownika na procesorze graficznym przyniesie wymierne korzyści. W pracy muszą znaleźć się badania potwierdzające, lub obalające tę tezę. Dodatkowo wszystkie badania muszą być przeprowadzone w warunkach naturalnych, gdyż do takich warunków projektowana jest aplikacja. Do każdych badań powinien być dołączony komentarz, a wyniki powinny być zaprezentowane w sposób przejrzysty i czytelny.

Dodatkowo w ramach pracy należy zaproponować i zaimplementować sposób umożliwiający integrację systemu z innymi programami. Ma to na celu dostarczenie wyników do innych programów. W ten sposób zapewni się rozszerzalność aplikacji i uniknie się hermetyzacji rozwiązania.

Podsumowując, wynikiem przeprowadzenia wyżej wymienionych zadań ma być optymalna aplikacja spełniająca wymagania interfejsu komputer-człowiek. Ponadto w pracy powinno znaleźć się omówienie badania potwierdzające założenia ideologiczne oraz wnioski prowadzące do stwierdzenia słuszności bądź niezgodności z poczynionymi krokami w celu stworzenia najlepszej implementacji systemu.

2.3 Przegląd pracy

W pierwszej części pracy zostały opracowane podstawy teoretyczne wymagane do zrozumienia pracy. W podrozdziale 3.1 znalazł się opis metod zrównoleglnia kodu oraz omówienie zagadnień z tym związanych. Następnie w podrozdziale 3.2 znalazł się opis przetwarzania obrazów uwzględniający jedynie metody użyte w pracy. Ostatni podrozdział podstaw teoretycznych dotyczy sieci komputerowych w zagadnieniach związanych z interfejsami.

Rozdział 4 to wyjaśnienie i opis rozwiązania. Składa się on z wymagań i założeń oraz

poszczególnych podrozdziałów dotyczących kolejno: algorytmów przetwarzania równolegiego (4.2), algorytmu wydzielania statycznego tła (4.3), algorytmu wydzielania dloni (4.4) oraz rozwiązań interfejsu sieciowego (4.5).

Na końcu pracy znajduje się podsumowanie, bibliografia oraz dodatki zawierające zrzuty ekranu (Dodatek A), kod źródłowy (Dodatek B) oraz opis użytych bibliotek programistycznych (Dodatek C).

Rozdział 3

Podstawy teoretyczne, przegląd literatury

Aby zrozumieć treść pracy, czytelnik powinien zapoznać się z podstawami teoretycznymi opisanymi w tym rozdziale. Ma to na celu wyjaśnienie zawartej w dalszych częściach treści oraz możliwość własnej oceny zaproponowanego sposobu na tle innych rozwiązań opisanych również w tym rozdziale. Rozdział porusza trzy podstawowe zagadnienia. Są to metody zrównoleglania kodu, analiza i przetwarzanie obrazów z szczególnym uwzględnieniem segmentacji tła oraz sieci komputerowe. W takiej kolejności tematy te są zaprezentowane i omówione poniżej.

3.1 Opis metod zrównoleglania kodu

W aplikacjach działających w czasie rzeczywistym w celu przyspieszenia działania wykorzystuje się optymalizację kodu. Często kosztem jakości obliczeń stosuje się algorytmy, których wyniki nie są tak dokładne, lecz rezultat otrzymywany jest w krótszym czasie. W pewnym momencie dochodzi się do bariery, której nie da się przekroczyć konwencjonalnymi metodami i program uznawany jest wtedy za optymalny.

W przypadku, gdy powyższe nie jest wystarczające dla bieżącego zastosowania, lub gdy wymagane jest wykonywanie dodatkowych operacji w tym samym czasie można skorzystać z przetwarzania równoległego. Często mylone w literaturze obliczenia współbieżące i rozproszone nie odnoszą się do tego samego pojęcia.

- Obliczenia współbieżące – są to procesy wykonywane w tym samym czasie, czyli innymi słowy kolejne rozpoczynają swoje wykonanie zanim poprzednie się zakończą. Obliczenia współbieżące mogą być realizowane nawet na procesorach jednordzeniowych poprzez zastosowania podziału czasu procesora pomiędzy poszczególne pro-

cesy. Obecne systemy operacyjne w większości wspierają obliczenia współbieżne, które są wymagane do działania wielu programów jednocześnie (choć fizycznie na maszynach jednoprocesorowych ich kod nie jest wykonywany w tym samym czasie).

- Obliczenia równoległe – to realizowane na wielu maszynach lub wielu procesorach taj samej maszyny obliczenia współbieżne
- Obliczenia rozproszone – są to procesy wykonywane w systemach, w których nie ma globalnej przestrzeni adresowej, tak więc wymiana informacji następuje poprzez wysyłanie komunikatów. Są to najczęściej wieloprocesorowe maszyny z pamięcią lokalną połączone w sieci komputerowe. Przykładem takich obliczeń jest SETI @ Home [12].

Podsumowując powyższe nie każdy współbieżny kod musi być wykonywany jednocześnie. Dopiero przy zastosowaniu fizycznej architektury wielowątkowej (to jest, gdy istnieje wiele potoków przetwarzania informacji) można mówić o przyspieszeniu działania kodu poprzez użycie zrównoleglenia.

Podczas wykonywania programu i testów używany był komputer z procesorem graficznym NVIDIA® Quadro 2000M, który dysponuje 192 wątkami przetwarzania danych.

3.1.1 Rozwój obliczeń równoległych na podstawie przetwarzania obrazów

Już w latach 50 próbowało dokonywać obliczeń równoległych dla algorytmów przetwarzania obrazów. Początkowo jednak nie istniał sprzęt komputerowy będący w stanie przetwarzać równolegle wielu danych na raz, dlatego też pierwsze architektury potrafiące przetwarzać wiele danych jednocześnie (ILLIAC IV, MPP, CLIP4, CLIP7) lub przetwarzające wiele danych używając wielu jednostek obliczeniowych (Cytocomputer, ZMOB) powstały dużo później [2]. Architektury procesorów zostały przedstawione w podrozdziale 3.1.6. Jeszcze w latach osiemdziesiątych zmagano się z brakiem wystarczającego zaplecza ekonomicznego czy technicznego by prawdziwie móc mówić o obliczeniach równoległych.

Rozwój algorytmów służących do przetwarzania obrazów doprowadził do stworzenia wielu rozszerzeń instrukcji procesorów oraz technologii istniejących do dziś w komputerach. Zgodnie z pracą [2] są to na przykład:

- MMX – Multimedia extension,
- SCREW (Visual Instruction Set) firmy SUN,
- MAX (Media ACCelerator) firmy HP,

- MVI (Video Motion Instructions) firmy DIGITAL,
- AltiVec firmy Motorola,
- SSE firmy Intel®,
- Wiele innych.

Dodatkowo, powszechność zastosowań przetwarzania multimediiów oraz kompresji danych skutkowała stworzeniem wielu koncepcji programistycznych jak i architektur zoptymalizowanych do rozwiązywania tego typu problemów.

W połowie lat dziewięćdziesiątych wprowadzono koncepcję gridu, czyli przetwarzania sieciowego. Jest to rozszerzenie koncepcji dzielenia się czasem obliczeniowym komputera z lat sześćdziesiątych [16]. Grid jest to system, który integruje i zarządza wieloma niejednorodnymi systemami udostępniającymi różne zasoby połączonymi przez sieć komputerową. Tworzą one razem jeden wirtualny komputer. Wymiana danych następuje przez otwarte i standardowe protokoły [15]. W technologiach gridowych obliczenia wykonywane są na bardzo wielu maszynach równocześnie. Pozwala to, przy pewnych założeniach dotyczących sieci, połączeń i niezawodności, na stworzenie systemu bardzo dobrze skalowanego, umożliwiającego przetwarzanie ogromnych partii danych w krótkim czasie. Przykładem gridu jest SETI @ home [16]. Systemy gridowe nie są używane przy tworzeniu systemów czasu rzeczywistego. W takich programach rozwiązania te nie są w stanie osiągnąć swych pełnych możliwości, jako że narzut na komunikacje dominuje w stosunku do zysku uzyskanego z szybszych obliczeń.

Potrzeba przetwarzania obrazów wraz z upowszechnieniem się grafiki trójwymiarowej skutkowała powstaniem procesorów dedykowanych tylko i wyłącznie do przetwarzania i wyświetlania grafiki. W 1975 roku IBM wprowadził na rynek pierwszy komputer osobisty wyświetlający grafikę – IBM 5100. Jednak termin GPU (Graphic Processing Unit) pojawił się dopiero w 1999 roku wraz z wprowadzeniem układu firmy NVIDIA® GeForce 256. Początkowo procesory graficzne odpowiedzialne były tylko za wyświetlanie grafiki, jednak ostatnimi czasy za pomocą specjalnych interfejsów programistycznych umożliwione zostało wykonywanie na nich praktycznie dowolnych operacji matematycznych.

Abstrahując od rozwiązań sprzętowych równolegle rozwijało się oprogramowanie. Wraz z powstaniem języków wysokiego poziomu pojawił się problem z brakiem rozwiązań umożliwiających korzystanie ze wszystkich zalet przetwarzania równoległego. Początkowo by uzyskać prawdziwie równoległy i optymalny kod programista był zmuszony do rozwiązywania problemów sekcji krytycznych, zakleszeń, zagłodzeń i wszystkich innych

związań z przetwarzaniem wielowątkowym. Problemy te są krótko wyjaśnione w kolejnym podrozdziale. By móc tworzyć wydajne aplikacje informatyczne zmuszeni byli więc wrócić do programowania niskopoziomowego, ponieważ tylko tak mogli czerpać wszelkie korzyści jakie dawała im powstała technologia. Jednak wraz z upowszechnieniem się i ujednoliceniem interfejsów danych oraz z narastającą powtarzalnością typowych zastosowań, powstawały efektywne rozwiązania wysokopoziomowe umożliwiające uzyskiwanie optymalnych rozwiązań nawet przy użyciu języków wyższego poziomu. Ujednolicenie stosowanych metod doprowadziło między innymi do powstania modeli spójności, opisanych w podrozdziale 3.1.3.

Zapotrzebowanie na tworzenie programów wykorzystujących przetwarzanie równoległe doprowadziło do powstania języków programowania specyficznych dla tych rozwiązań. Na poziomie takiego języka znajdowało się wsparcie dla współbieżności, przekazywania komunikatów czy obsługi pamięci współdzielonej. Takimi językami były na przykład Erlang, Ada, Pict, Limbo czy occam oraz wiele innych. Niektóre z nich jak na przykład Erlang są używane do dziś w zastosowaniach w przemyśle [3]. Obecnie znane i powszechnie wykorzystywane języki wspierające przetwarzanie równoległe to na przykład: Java i C#.

Wraz z powstaniem języków programowania równoległego ważnym okazało się ujednolicenie oraz stworzenie hierarchizacji powszechnych problemów i sposobów ich rozwiązywania. Aktualizacja i synchronizacja danych jest niewątpliwie najważniejszym zagadnieniem przy programowaniu równoległym. Dlatego też powstały modele spójności, które są szczegółowo opisane w podrozdziale 3.1.3.

Rozwój języków programowania doprowadził do powstania bibliotek programistycznych zawierających typowe procedury, które mogły być używane przez wielu programistów do podobnych zastosowań. Między innymi powstały biblioteki do przetwarzania obrazów jak i obliczeń równoległych. Użycie takiej biblioteki umożliwia szybsze pisanie kodu i osiągnięcie zamierzzonego efektu. W pracy została użyta biblioteka OpenCV oraz rozwiązanie NVIDIA® CUDA™. Biblioteki te opisuje Dodatek C.

3.1.2 Podstawowe problemy przetwarzania równoległego

Poniżej zostały opisane typowe problemy występujące przy przetwarzaniu równoległym.

Zagłodzenie

Sytuacja w środowisku wielozadaniowym, w której teoretycznie proces może ukończyć swoje przetwarzanie, ale nie otrzymuje wymaganego czasu procesora lub innych zasobów.

Zagłodzenie występuje najczęściej na skutek błędnej implementacji algorytmu rozdzielania zadań, lub w przypadku nadmiernego obciążenia systemu.

Zakleszczenie

Jest to sytuacja, gdy w środowisku wielozadaniowym przynajmniej dwa równocześnie wykonujące się procesy czekają na siebie nawzajem. Sytuacja taka może mieć miejsce gdy wymagane zasoby są zajmowane w różnej kolejności. Jest to częsty błąd podczas programowania równoległego skutkujący przeważnie zawieszaniem działania aplikacji.

Do zakleszczenia dochodzi gdy system spełnia następujące warunki:

- Występuje wzajemne wykluczanie - w danej chwili z zasobu może korzystać jedynie jeden proces.
- Cykliczne oczekiwanie - zadania żądają zasobów w sposób, że powstaje cykliczny graf zależności (w przypadku gdy wszystkie procesy zajmują zasoby w tej samej kolejności nie dojdzie do zakleszczenia).
- Trzymanie zasobu i oczekiwanie - blokowany jest kolejny zasób przed odblokowaniem poprzedniego (często jest to niezbędne w celu zachowania spójności danych).
- Brak wywłaszczenia z zasobu - proces nie zwalnia zasobu przed zakończeniem wykonywania swoich działań.

Aby uniknąć zakleszczenia, wystarczy doprowadzić do sytuacji, w której nie jest spełniony przynajmniej jeden z wyżej wymienionych warunków. Zakleszczenie występuje najczęściej na skutek błędnego projektowania aplikacji.

3.1.3 Modele spójności

Wraz z rozwojem obliczeń równoległych powstał problem z przesyłaniem danych pomiędzy jednostkami zajmującymi się obliczeniami. Potrzebne okazało się ujednolicenie zagadnień z tym związkanych. Spowodowało to powstanie zunifikowanych modeli spójności.

W systemie rozproszonym dane zmieniane w jednym węźle muszą zostać rozpropagowane. Wiąże się to z komunikacją i opóźnieniami, dlatego mogą występować niespójności, co nie jest pożądane. Modele spójności określają sposób w jaki te dane będą aktualizowane. Istnieje wiele poziomów spójności danych, w zależności od potrzeb aplikacji.

Najbardziej restrykcyjna jest spójność ścisła. Oznacza ona, że każdy odczyt zmiennej zwraca wartość zaktualizowaną po ostatnim wykonanym zapisie tej zmiennej. Często

jednak w systemie zdecentralizowanym trudno określić, która operacja jest ostatnia, ponieważ nie ma wspólnego zegara. Spójność taka jest rzadko stosowana, gdyż w praktyce oznacza, że system traci właściwości systemu zdecentralizowanego.

Modele spójności można podzielić na dwie grupy, są to:

- Modele spójności nastawione na dane.
- Modele spójności nastawione na klienta.

Te pierwsze zakładają, że dane są związane są z jednym serwerem. W trakcie ich przetwarzania nie następuje ich migracja, dlatego realizacja spójności jest prostsza. Jeżeli jednak dopuszcza się by dane mogły ulegać migracji należy zadbać o implementację modelu spójności nastawnionego na klienta.

W pracy nie było potrzeby analizy modelów spójności, ponieważ synchronizacja danych następowała każdorazowo po jakiekolwiek operacji. Dzięki temu jednolitość danych była zachowana w trakcie całego przetwarzania obrazów z kamery. Ponadto, w ramach wykonywania pojedynczej operacji na obrazie dane zostają odseparowane, a przetwarzanie każdej z części danych dokonywane jest niezależnie.

3.1.4 Prawo Amdahla

Maksymalne przyspieszenie wykonywania kodu jest uzależnione głównie od możliwości zrównoleglenia tego kodu. Gene Myron Amdahl sformułował prawo dotyczące maksymalnego wzrostu szybkości działania aplikacji dzięki zrównolegleniu jej kodu.

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}} \quad (3.1)$$

Wzór 3.1 pokazuje maksymalny wzrost wydajności wykonania kodu równoległego w stosunku do kodu sekwencyjnego. $S(N)$ oznacza maksymalny wzrost przyspieszenia działania aplikacji. P jest to część programu, która może ulec zrównolegleniu podana w procentach stosunku czasu wykonania do całego programu. N oznacza liczbę strumieni przetwarzania kodu.

Z powyższego wzoru widać, że nie jest możliwe przyspieszanie działania aplikacji w nieskończoność poprzez dodawanie kolejnych wątków. Dodatkowo bardzo duże znaczenie ma ilość kodu, który nie ulega zrównolegleniu (czyli na przykład przechwytcie informacji z urządzenia, operacje na wszystkich danych jednocześnie). Jak zostało pokazane w późniejszych podrozdziałach (4.2.3) mimo posiadania nawet ponad stu wątków przyspieszenie działania aplikacji jest co najwyżej kilkukrotne.

Warto również zauważyć, że kod wykonywany na wielu procesorach (część wykonywana równolegle) ulega łatwiejszemu skalowaniu. Przy przetwarzaniu obrazów przekłada

się to na możliwość zwiększenia rozdzielczości przetwarzanych obrazków przy zachowaniu podobnych czasów wykonania programu.

3.1.5 Rodzaje zrównoleglania kodu

Algorytmy, w których występuje równoległość charakteryzują się istniejącymi rozgałęzieniami w trakcie wykonywania kodu, po których przetwarzanie jest niezależne. Rozgałęzienia takie dzielimy na:

- Schodzące się do jednego punktu po zakończeniu przetwarzania, czyli innymi słowy są to instrukcje, w których liczba instrukcji do przetworzenia jest ustalona, a czas ich wykonania jest deterministyczny
- Nieschodzące się, ale w których występuje synchronizacja obliczeń związana z wymianą informacji. W takim podejściu obliczenia równolegle są realizowane, aż do osiągnięcia pewnego warunku zbieżności i dopiero wtedy następuje koniec przetwarzania. Czas wykonania takich instrukcji nie jest deterministyczny.

Programowanie równolegle niesie ze sobą zwiększenie kosztu wytwarzania oprogramowania. Istnieją trzy realizacje obliczeń równoległych w programie. Są to:

- Zrównoleglenie na poziomie kodu – istniejący kod zostaje zrównoległy w miejscach gdzie jest to możliwe poprzez użycie automatycznych narzędzi zrównoleglania kodu. Programista nie zmienia koncepcji rozwiązania, lecz jedynie oznacza miejsca w programie, które będą przetwarzane równolegle. Metoda ta wymaga najmniejszego nakładu pracy jednak jest również mniej efektywna jeżeli chodzi o uzyskane wyniki.
- Zrównoleglenie na poziomie metody – zimplementowanie istniejącego zadania z uwzględnieniem metod, czyli algorytmów, które nadają się do użycia równoległego przetwarzania. Koncepcja całego zadania zostaje ta sama, jednak poszczególne części programu są przepisane.
- Zrównoleglenie na poziomie zadania – całe zadanie jest zaprojektowane od początku z myślą o przetwarzaniu równoległym. Najlepsza metoda pod względem wydajnościowym, lecz zarazem najtrudniejsza i wymagająca największego nakładu pracy.

Rozwiązanie zastosowane w programie należy do zrównoleglania na poziomie metody gdyż cała koncepcja przetwarzania pozostała ta sama, jednak zmienione zostały poszczególne metody rozwiązywania etapów przetwarzania. Możliwe to było ponieważ pro-

jektując aplikację od początku była ona tworzona w sposób umożliwiający na późniejsze rozszerzenie i zaimplementowanie przetwarzania równoległego. Dodatkowo istniejące rozwiązanie zostało już dokładnie przebadane pod względem jakości działania, co zdecydowanie zmniejszało ilość wymaganej pracy. Ponadto przeanalizowano inne możliwości zrównoleglenia na poziomie zadania i nie znaleziono żadnego, dającego wymiernie lepsze korzyści, rozwiązania.

3.1.6 Taksonomia Flynn'a

W latach 60, długo przed upowszechnieniem się wielordzeniowych procesorów Michael Flynn zaproponował podział architektur komputerowych ze względu na liczbę przetwarzanych strumieni instrukcji jak również danych. W taksonomii tej zostały wyróżnione cztery grupy:

- SISD (Single Instruction, Single Data) – pojedynczy strumień instrukcji, pojedynczy strumień danych – w tej grupie znajdują się komputery skalarne. Pierwsze komputery, w których procesory były jednordzeniowe – pierwsze popularne PC'ty należą do tej grupy.
- SIMD (Single Instruction, Multiple Data) – tak jak poprzednio jest jeden strumień przetwarzania instrukcji, lecz wiele strumieni danych używanych do wykonywania programu. Do tej grupy należą komputery wektorowe, gdzie te same operacje należy wykonać na wielu różnych danych. Również do tej grupy należy zaliczyć procesory wielowątkowe jak i procesory graficzne (GPU).
- MISD (Multiple Instruction, Single Data) – kolejna grupa jest bardzo rzadko spotykana. Należą do niej maszyny, w których wiele programów wykonuje operację na tych samych danych. Można przyjąć, że zastosowania takie służą wyłącznie dla zwiększenia bezpieczeństwa poprzez redundantne wykonanie różnych metod obliczeniowych na tych samych danych w celu zminimalizowania ryzyka błędów. Zastosowania tej architektury można znaleźć na przykład w pojazdach kosmicznych gdzie błędy są bardzo kosztowne.
- MIMD (Multiple Instruction, Multiple Data) – wykonywane jest wiele programów jednocześnie, gdzie każdy przetwarza własny strumień danych. Przykładem takiego zastosowania są superkomputery wieloprocesorowe, klastry lub gridy.

Dla potrzeb analizowanego w tej pracy problemu wykorzystywana jest architektura SIMD. Został stworzony jeden program uruchamiany na wielu wątkach procesora graficznego. Dane zostały podzielone równomiernie w celu przyspieszenia uzyskania wyniku końcowego.

| | Pojedynczy strumień instrukcji | Wiele strumieni instrukcji |
|----------------------------|--------------------------------|----------------------------|
| Pojedynczy strumień danych | SISD | MISD |
| Wiele strumieni danych | SIMD | MIMD |

Tabela 3.1. Taksonomia Flynnna.

Istnieje również pojęcie SPMD czyli Single Process Multiple Data, co oznacza pojedynczy proces wiele danych. Jest to również technika używana do osiągnięcia równoległości. Jest to podtyp MIMD z tym, że wykonywane instrukcje tworzą jedną, tą samą procedurę dla różnych danych. SPMD różni się od SIMD tym, że wiele różnych procesorów wykonuje ten sam program, ale nie koniecznie wykonując te same instrukcje jednocześnie. Przeważnie używa się tego pojęcia do programów używających komunikatorów i rozproszonej pamięci [7].

3.1.7 Programowanie równoległe na procesorach wielordzeniowych

W celu stworzenia programu, który wykorzystuje architekturę wieloprocesorową programista jest zmuszony do użycia mechanizmów zapewniających uruchamianie kodu działającego współbieżnie. By uzyskać ten efekt możliwe jest wykorzystanie wielu narzędzi takich jak tworzenie i zarządzanie wieloma procesami lub wątkami ręcznie używając do tego jedynie wsparcia systemu operacyjnego i funkcji jakie są wbudowane w języki programowania. Innym rozwiązaniem jest skorzystanie z bibliotek programistycznych zapewniających wysoko poziomowe API do zarządzania równoległym kodem.

Istnieje wiele bibliotek programistycznych wspierających programowanie równoległe zarówno na wielordzeniowych procesorach CPU jak i procesorach graficznych. Cel użycia i charakter aplikacji równoległych często znacząco się różni, dlatego powstało wiele rozwiązań wspierających pisanie takich programów. Poniżej zostało przedstawione kilka popularnych obecnie rozwiązań.

- OpenMP [11] – wieloplatformowy interfejs służący do programowania współbieżnego programów działających na maszynach wieloprocesorowych z dzieloną pamięcią wspólną. Jest to zestaw bibliotek programistycznych wraz z zestawem zmiennych środowiskowych i dyrektyw kompilatora umożliwiających zrównoleglenie kodu przez dodanie nielicznych instrukcji do już istniejących źródeł programu. OpenMP charakteryzuje się przenośnością oraz faktem, że jest dostępny na wiele platform. Rozwiązanie umożliwia zrównoleglenie kodu jedynie dla procesora CPU, tak więc nie zostało brane pod uwagę przy tworzeniu programu.

- Intel® MKL [13] – biblioteka programistyczna służąca głównie do zastosowań matematycznych i finansowych. Zawiera zestaw zoptymalizowanych algorytmów z zakresu zaawansowanej matematyki pod najnowsze procesory CPU. Tak jak powyższe rozwiązanie Intel MKL posiada wsparcie jedynie dla jednostek centralnych i pomija rozwiązywanie zadań przy użyciu procesora graficznego.
- OpenCL (Open Computing Library) [14] – platforma programistyczna wspierająca tworzenie programów na heterogenicznych platformach. Pozwala na implementację systemów zarówno na CPU jak i na procesory graficzne. Została zapoczątkowana przez firmę Apple Inc. OpenCL jest bardzo podobna do rozwiązania NVIDIA® CUDA™ z tą różnicą, że jest wieloplatformowa dzięki czemu możliwe jest również wykonywanie kodu napisanego w tej technologii na procesorach graficznych firmy ATI/AMD. Cechą negatywną OpenCL jest fakt, że nie posiada ona wsparcia dla technik specyficznych dla danego sprzętu z racji wcześniej wspomnianej wieloplatformowości. Platforma OpenCL nie została użyta podczas pisania pracy. Jest to związane z użyciem biblioteki OpenCV, która posiada wsparcie jedynie dla rozwiązania firmy NVIDIA®. Dodatkowym argumentem przemawiającym przeciwko OpenCL jest wygoda programowania (na korzyść CUDA™).
- NVIDIA® CUDA™ [6] – opracowana przez firmę NVIDIA® technologia wspierająca obliczenia równolegle na procesorach graficznych. W przeciwieństwie do poprzedniego rozwiązania CUDA™ posiada wsparcie jedynie dla kart graficznych NVIDIA®. Rozwiązanie oferuje wsparcie dla rozwiązań astrofizycznych, biologicznych, chemicznych, z zakresu dynamiki płynów, elektromagnetycznych i wielu innych. Dodatek C zawiera obszerne opisanie rozwiązania CUDA™. Poniżej znajduje się kilka przytoczonych najważniejszych informacji.

Rozwiązanie CUDA™ zostało wprowadzone w 2007 roku i obecnie dostępna jest już wersja 4.1. Biblioteka jest rozpowszechniona i chętnie używana przez tysiące programistów. Została ona dedykowana dla języka C, C++, ale istnieje wiele rozszerzeń umożliwiających używanie jej zalet w językach Fortran, Java, Python, .NET i wielu innych. Podobnie jak OpenCV, CUDA™ jest wieloplatformowa (Linux, MacOS, Windows).

CUDA™ implementuje dwie bardzo znane biblioteki obliczeniowe BLAS i FFT odpowiednio cuBLAS i cuFFT, jak również bibliotekę CUDPP (Data Parallel Primitives) oferujących algorytmy takie jak równoległą redukcję czy sortowanie.

Do działania technologii CUDA™ wymagany jest osobny kompilator nvcc. Jest to kompilator potrafiący skompilować kod C jak i dyrektywy procesora graficznego. Kompilator ten posiada zdolność komplikacji do kodu C (wykonawanego na CPU) jak również

obiektów wykonywanych na GPU. Pliki wykonywalne wymagają biblioteki CUDATM oraz biblioteki uruchomieniowej *cudart*.

Technologia CUDATM oferuje skalowalny model programowania. Model ten opiera się na trzech podstawowych pojęciach. Są to grupy wątków, wspólna pamięć oraz synchronizacja za pomocą bariery. Operując na tych abstrakcjach programista tworzy równoległy kod o zdekomponowanej strukturze. W praktyce oznacza to, że programista nie zajmuje się tym na ilu wątkach będzie działać program, gdyż jest to transparentne na poziomie tworzenia kodu. Dopiero w trakcie uruchamiania programu determinowana jest liczba fizycznych wątków przetwarzania. Oznacza to, że programy mogą zostać uruchamiane na bardziej wydajnych kartach graficznych bez zmiany koncepcji oraz ingerencji w kod źródłowy [6].

Opisane powyżej usprawnienia programistyczne nie są jedynymi istniejącymi, jednak zostały tutaj wymienione z powodu ich popularności i solidności wykonania. Są to różne rozwiązania posiadające odmienne cechy i służące do odmiennych zastosowań. W pracy została użyta biblioteka NVIDIA® CUDATM wraz z biblioteką OpenCV, która zawiera również implementację algorytmów na procesor graficzny.

Odmienne wykorzystanie procesorów graficznych jest wykorzystywane w grach i grafice komputerowej. Zastosowanie to jest pierwotną przyczyną powstania procesorów graficznych i służy głównie podczas renderowania scen trójwymiarowych w jak najkrótszym czasie. Dla takiego, nominalnego użycia procesorów graficznych używane są rozwiązania inne niż powyższe. Są to biblioteki takie jak DirectX czy OpenGL. W ostatnich czasach procesorów graficznych zaczęto również używać do symulacji fizyki (głównie w grach komputerowych). Służy do tego na przykład biblioteka NVIDIA® PhysX.

3.1.8 Przykładowe programy czerpiące korzyść ze zrównoleglenia kodu

Dla pełnego zrozumienia korzyści zrównoleglania kodu przeanalizowano kilka istniejących rozwiązań. Poniżej zaprezentowano wyniki niektórych prac.

W pracy [9] rozwiązywano wymagający obliczeniowo problem poprawy jakości wideo przez badanie i usuwanie pionowych zadrapań w sekwencji wideo. Zastosowano algorytm genetyczny, który okazał się być bardzo dobrze skalowalny przy zastosowaniu obliczeń równoległych, dzięki czemu udało się osiągnąć dobre wyniki przyspieszenia działania aplikacji. Autorzy zaproponowali podział problemu na pojedyncze zadrapania jak również na poszczególne wiersze obrazu. W ten sposób osiągnęli bardzo wiele niezależnych obliczeniowo zadań. W przypadku gdy pojedynczych zadań było więcej niż dostępnych wątków obliczeniowych, zostawały one dzielone równo pomiędzy poszczególne wątki. Dla zadanego testu udało uzyskać się 2,7 krotne przyspieszenie (z 1574 sekund do 565 sekund).

Artykuł [18] traktuje o technologii umożliwiającej równoległe przetwarzanie pikseli. W pracy brane jest pod uwagę zarówno zrównoleglenie przetwarzania danych jak również zrównoleglenie metody. Zaproponowane rozwiązanie opiera się na podziale pikseli obrazu na poszczególne „koszyki”, które mogą zostać przetwarzane niezależnie przez pojedyncze wątki. Rozwiązanie opiera się na założeniu, że do przetwarzania brane są specjalnie wyselekcjonowane piksele z obrazu wejściowego. Sposób przypomina obliczenia na macierzach rzadkich, jednak jest on specyficzny dla algorytmów działających na obrazach. Autorzy wyodrębnili trzy poziomy przetwarzania obrazów:

- Niskopoziomowe - oparta na samym obrazie, przeważnie działające jedynie na pojedynczych pikselach. Oferują bardzo dobry stopień zrównoleglenia. Do realizacji równoległej używa się przeważnie architektury SIMD. Przykładem może być operacja rozmycia.
- Średniopoziomowe - oparte na symbolach, dotyczące pikseli należących do danego obiektu. Dla przykładu śledzenie obiektu. Dla realizacji wielowątkowej przeważnie używa się architektury SPMD jednakże może być ona zastąpiona SIMD lub MIMD.
- Wysokopoziomowe - oparte na bazie wiedzy, a więc algorytmów wykorzystujących informacje nie tylko o obrazie, ale również dotyczących samego zagadnienia przetwarzania. Typowo dla takich zastosowań najlepiej wykorzystać architekturę MIMD.

Biorąc pod uwagę powyższe autorzy stworzyli szkielet na bazie którego możliwe było równoległe przetwarzanie algorytmów operujących na obrazach.

W zupełnie inny sposób jest rozwiązany problem przetwarzania równoległego w pracy [5]. Zamiast obliczeń dokonywanych na jednej maszynie z wielowątkowym procesorem lub procesorem graficznym w systemie zastosowano trzy niezależne maszyny. W pracy rozważano rekonstrukcję postury ludzkiej i przełożenie jej na wirtualną kukłę za pomocą widzenia stereoskopowego z dwóch kamer. Pierwsze dwa komputery przetwarzaly zupełnie niezależnie obraz dla pojedynczych kamer, a trzeci komputer składał wyniki i analizował przetworzone wstępnie informacje z poprzednich maszyn. Komunikacja pomiędzy komputerami następowała za pomocą komunikatów wysyłanych przez sieć TPC/IP. System został zaprojektowany w oparciu o model człowieka do którego były dopasowywane dane z kamer w celu uzyskania realistycznego i odpornego na zakłócenia ruchu.

3.1.9 Podsumowanie

Dla wielu zastosowań obecne procesory i przetwarzanie sekwencyjne jest za wolne. Szczególnie, jeśli chodzi o systemy czasu rzeczywistego, w których wymagana jest jak najwięks-

sza wydajność. Przy przetwarzaniu obrazów przeważnie możliwa jest dekompozycja obrazu na pojedyncze piksele, dzięki czemu umożliwione zostaje przetwarzanie równoległe różnych danych poprzez różne wątki. Dla wielu zastosowań wykorzystanie procesorów graficznych (jak również każdego innego systemu SIMD lub MIMD) pozwala na uzyskanie lepszych wyników wydajnościowych.

Biorąc pod uwagę wyniki uzyskane podczas zrównoleglania działania aplikacji napisanej wraz z tą pracą uzyskuje się nie tylko przyspieszenie działania aplikacji, ale również odciąża się główny procesor, zrzucając zadania na procesor graficzny, który w większości czasu nie wykonuje żadnych obliczeń. Podczas gdy wykonywana aplikacja ma być jedynie pomocniczą aplikacją interfejsu użytkownika zaleta ta powoduje szybsze działanie systemu operacyjnego, a co za tym idzie płynniejsze działanie całego komputera. Więcej o wynikach zrównoleglenia można przeczytać w podrozdziale 4.2.3.

3.2 Przetwarzanie obrazów

W pracy nie tylko badano możliwość zrównoleglania kodu na podstawie aplikacji rozpoznawania gestów i ruchu dloni, ale również skupiono się na poprawieniu jakości jej działania. W tym celu zastosowano autorski algorytm substrakcji tła, który został opracowany na podstawie obserwacji i inspiracji innymi algorytmami. Zakładając, że aplikacja będzie używana ze statycznie umiejscowioną kamerą algorytm taki daje znakomite poprawienie jakości w funkcjonowaniu programu. Dodatkowo w pracy zastosowano optymalizację algorytmów przetwarzania obrazów w celu poprawienia czasu analizy danych z kamery.

Podrozdział ten jest omówieniem podstaw z zakresu przetwarzania obrazów. Dodatkowo zostały tu omówione konkurencyjne algorytmy służące do segmentacji tła jak i kilka innych prac dotyczących przetwarzania obrazów.

Kolejne części opisują algorytmy użyte w przetwarzaniu obrazów w końcowej implementacji systemu. Skupiono się na tych, które zostały zaimplementowane umożliwiając ich wykonanie w sposób równoległy.

Na początku zostały opisane podstawowe algorytmy przetwarzania obrazów, a następnie szczególna uwaga została zwrócona na algorytmy wydzielania tła. Są one bardziej zaawansowane i dlatego istnieje wiele znaczco różniących się rozwiązań. Na podstawie literatury przedstawiono część z tych najbardziej popularnych.

3.2.1 Podstawowe algorytmy przetwarzania obrazów

W tej części zostały omówione niektóre podstawowe algorytmy przetwarzania obrazów. Skupiono się na tych używanych w niniejszej pracy. Opisane zostały jedynie podstawy,

więcej na ten temat można przeczytać w pracy [22] oraz w książce [19].

Algorytmy punktowe

Operacje, które przekształcają piksele obrazu niezależnie od pozostałych pikseli, nazywane są operacjami punktowymi. Przekształcenia te, mogą w znaczący sposób zmienić wygląd obrazu, zarówno poprawiając jego jakość i uwidaczniając interesujące cechy.

Operacje punktowe, z racji swojego charakteru, pozwalają na bardzo szybkie ich wykonanie. Złożoność algorytmów realizujących przekształcenia punktowe jest liniowa ($O(n)$). Dodatkowo możliwe jest wykonywanie operacji punktowych równolegle dla wielu pikseli, co przy wielordzeniowości dzisiejszych procesorów graficznych przynosi znaczący zysk na czasie wykonania.

Przykładowe operacje punktowe to rozjaśnienie, utworzenie negatywu czy przyciemnianie. Inne bardziej zaawansowane to na przykład: selektywna zmiana kolorów, zmiana luminescencji poszczególnych kanałów, zaciemnianie kolorów części obrazu i tym podobne.

Do tego typu przetwarzania należą również operacje logiczne oraz algorytmy konwersji przestrzeni kolorów.

Przekształcenia kontekstowe

Operacje, dla których danymi wejściowymi nie są pojedyncze piksele, ale zbiory pikseli, nazywane są przekształceniami kontekstowymi, lub przekształceniami przy pomocy filtrów. Końcową wartość piksela oblicza się na podstawie jego poprzedniej wartości oraz informacji o sąsiednich pikselach.

Filtry z racji wymogu przeliczania dużej ilości informacji dla każdego pojedynczego piksela, są złożone obliczeniowo. Wymagają przez to znacznie więcej zasobów niż operacje punktowe omówione w poprzednim podrozdziale. Przy badaniu dużego obszaru piksela wejściowego spowolnienie może być nawet kilkudziesięciokrotne. Jednak również i te operacje, podobnie jak punktowe, mogą być wykonywane niezależnie od wyników dla kolejnych pikseli, co umożliwia ich dekompozycję i przyspieszenie wykonywania na procesorach wielordzeniowych.

Przekształcenia kontekstowe w znaczący sposób wpływają na przetwarzany obraz, przekształcając go również geometrycznie. Za ich pomocą można usuwać szum, jak również podkreślać krawędzie, lub inne słabiej widoczne informacje zawarte w obrazie. W literaturze operacje kontekstowe omówione są bardzo szeroko ze względu na ich elastyczność i możliwość dopasowania do wielu zastosowań. Większość operacji kontekstowych opiera się na użyciu maski filtra.

Maska jest najczęściej kwadratem zawierającym wagę pikseli obszaru, który będzie rozpatrywany w trakcie obliczenia wartości końcowej punktu obrazu. Typowy rozmiar maski to 3 na 3 pikseli z punktem relacyjnym na środku. Przy użyciu operacji kontekstowych występuje jednak problem z pikselami znajdującymi się na krawędzi obrazu. Dla tych pikseli nie jest możliwe dokładne obliczenie wyjściowej wartości. Mimo to przekształcenia kontekstowe są najczęściej i najczęściej używane w systemach rozpoznawania obrazów.

Za pomocą przekształceń kontekstowych można wykonać operacje takie jak wyostrzanie, rozmazywanie (usuwanie szumu), detekcja krawędzi i wiele innych.

Przekształcenia morfologiczne

Przekształcenia morfologiczne są to operacje działające na obszarze obrazu używane głównie w celu usunięcia znieksztalceń, zakłóceń i znieksztalceń powstały w wyniku segmentacji. Różnią się od przekształceń kontekstowych tym, że do ich obliczeń sprawdzany jest warunek logiczny, który musi zostać spełniony przez wartość pikseli z maski. Najbardziej znanymi przekształceniami morfologicznymi jest operacja erozji i dylatacji.

Operacje morfologiczne operują na maskach, zwanych inaczej elementami strukturującymi, czyli innymi słowy sąsiedztwie badanego punktu. Na podstawie zawartości znajdujących się wewnątrz maski pikseli obliczana jest wartość maksymalna bądź minimalna. Wyróżniamy dwie główne operacje morfologiczne, są to:

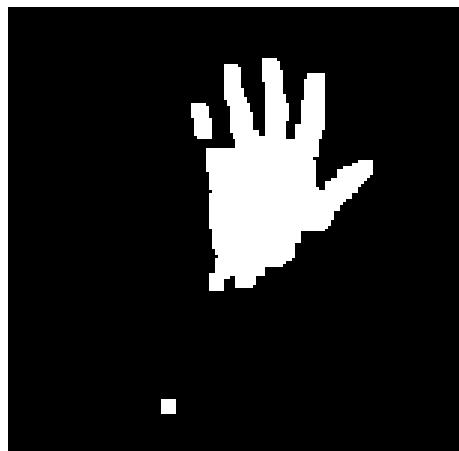
- Erozja – punkt wyjściowy uzyskuje wartość najmniejszą spośród pikseli znajdujących się wewnątrz elementu strukturującego.
- Dylatacja – w przeciwnieństwie do erozji, jako piksel wyjściowy brany jest piksel o wartości największej spośród pikseli znajdujących się wewnątrz maski.

By uzyskać oczekiwany efekt (na przykład usunąć zakłócenia ziarniste) powyższe operacje wykonuje się w parach. Jeżeli pierwszą z nich jest erozja, a następna dylatacja operacje taką nazywa się otwarciem. Kolejność odwrotna, czyli najpierw dylatacja, a następnie erozja to operacja zamknięcia. Operacji otwarcia używa się, gdy występuje potrzeba usunięcia drobnych punktów w obrazie, a operacji zamknięcia na przykład gdy obraz po segmentacji nie jest kompletny (występują dziury, niedoskonałości) i należy go wypełnić.

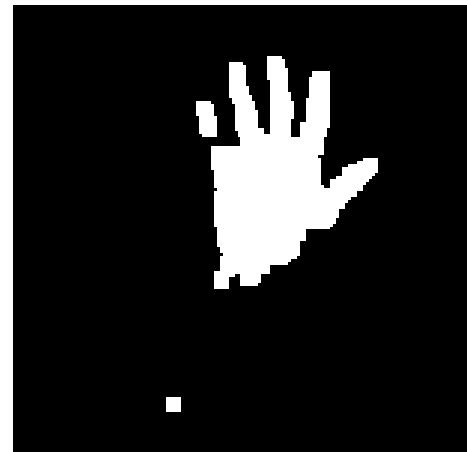
W pracy używana była głównie morfologia binarna dla kwadratowego elementu strukturującego. Oznacza to, że piksele wejściowe jak i wyjściowe mogły przyjmować jedynie wartości binarne (białe lub czarne). Dla takiego algorytmu możliwa była optymalizacja, która została opisana w podrozdziale 4.2.2.



Schemat 3.1. Obraz wejściowy.



Schemat 3.2. Obraz po erozji.



Schemat 3.3. Obraz po erozji i po dylatacji.

Na rysunkach 3.1 - 3.3 przedstawiono przykładowe działanie operacji zamknięcia oraz otwarcia. Rysunki przedstawiają kolejno erozję i dylatację użyte operacje w celu usunięcia zbędnych obszarów powstały po segmentacji skóry.

3.2.2 Algorytmy segmentacji tła

Wydzielanie tła z obrazu jest bardzo pomocne podczas rozpoznawania obiektów na obrazie. Wydzielenie i usunięcie obszaru niebędącego wyszukiwanym obiektem pozwala znacznie poprawić funkcjonowanie każdego algorytmu rozpoznawania obiektów na obrazach. Obecnie w zdecybowanej większości programów zajmujących się przetwarzaniem obrazów implementowany jest taki algorytm. W zależności od zastosowań istnieje wiele sposobów na wydzielanie tła zarówno ze statycznego jak i z dynamicznego obrazu (sekwencji obrazów). Poniżej przedstawiono kilka stosowanych w literaturze rozwiązań oraz

przeanalizowano możliwość ich zastosowania w tworzonym programie.

W pracy [1] zastosowano nietypowe podejście polegające na stworzeniu trójwymiarowego modelu pomieszczenia, w jakim znajduje się kamera i dynamicznej próby dopasowania tła do otaczającego terenu. Opisany algorytm odznaczał się większą niezawodnością szczególnie w przypadku, gdy kamera jest ruchoma. Algorytm ten okazał się również mniej podatny na zmianę luminescencji obrazu. Wyniki wydajnościowe osiągnięte podczas badań pomimo, że imponujące (20 klatek/s na 1.86 GHz AMD Athlon PC przy rozdzielcości 320 na 240) są jednak zdecydowanie niewystarczające dla programu, którego głównym zadaniem jest zastąpienie interfejsu użytkownik-komputer. Dodatkowo warto zaznaczyć, że w tworzonej pracy algorytm segmentacji tła jest jedynie częścią składową, a nie głównym celem, dlatego algorytm ten musi być również szybki i w miarę prosty.

Artykuł [23] przedstawia metodę segmentacji tła i elementów pierwszego planu operując na założeniu, że tło posiada podobny rozkład kolorów i barw. Wykorzystany algorytm nie służy do badania jedynie pojedynczych pikseli w celu klasyfikacji tło-nie tło, lecz poddawane analizie są piksele znajdujące się w sąsiedztwie danego punktu. Pozwala to na uzyskanie lepszych wyników w stosunku do algorytmu bez tej zmiany. W związku z faktem, że tworzony wraz z tą pracą program powinien działać niezależnie od tła na jakim segmentowana będzie dłoń algorytm ten nie mógł mieć zastosowania w tej pracy.

Praca [8] omawia kilka algorytmów wydzielania tła z obrazu w sekwencji wideo. W przeciwnieństwie do poprzednich prac omawiane tu rozwiązania zakładają działanie aplikacji w czasie rzeczywistym. Skutkuje to wymogiem, że ich wydajność musi być odpowiednio wyższa. Główna ideą rozwiązań jest badanie różnicy pomiędzy wieloma poprzednimi ramkami sekwencji wideo, a obecną. Jeżeli piksel jest odmienny od odpowiadającego mu piksela ze zbudowanego wcześniej modelu, jest traktowany jako nie należący do tła. By usprawnić działanie algorytmu w pracy [8] model danego piksela jest tworzony na podstawie statystyki występowania kolorów. Im dany kolor występuje częściej tym prawdopodobieństwo, że jest to kolor tła jest wyższe.

Samo dopasowanie aktualnego obszaru do obszaru tła lub ruchomego obiektu nie jest wystarczające dla poprawne i niezawodnie działającego algorytmu segmentacji tła. Innym problemem jest uaktualnianie modelu tła. Czasami pobieranie tła jest dokonywane tylko i wyłącznie na życzenie użytkownika, a następnie model ten jest zachowywany niezmienne. Rozwiązanie takie jest jednak pełne wad, ponieważ nie tylko w tle pojawiają się niechciane obiekty, ale również w przestrzeni czasu zmieniać się może luminescencja otoczenia. W pracy [8] zaprezentowano kilka różnych podejść rozwiązań powyższego problemu. Zostały one opisane poniżej.

Jedną z możliwych metod aktualizacji tła jest zamiana losowo wybranych pikseli na

piksele z aktualnego tła. Metoda taka generuje jednak niepotrzebne zakłócenia powodując zniekształcenia segmentowanych obiektów nie należących do tła. Co więcej, jeżeli obiekt nienależący do tła będzie poruszał się nieznacznie, bądź pozostało dłużej w bezczynności zostanie on zaklasyfikowany jako tło. Dodatkowo po takiej błędnej klasyfikacji mimo ruchu obiektu obszar, w którym się znajdował poprzednio będzie uznawany jako obiekt, mimo iż będzie tłem. Mówimy wtedy o błędnie pozytywnym rozpoznaniu.

Inna metoda – również przedstawiona w pracy [8] bazuje na ‘narastającym zboczu’. Im dłużej piksel pozostaje bez zmian tym bardziej prawdopodobne, że jest to tło. W tym algorytmie musi istnieć próg czasowy, po którym piksel zostaje uznany jako tło. Tak jak w poprzednim rozwiążaniu i tu występuje problem, gdy wydzielany obiekt przez dłuższy okres pozostaje bez ruchu. Po pewnym czasie zostanie on błędnie zaklasyfikowany jako tło.

Biorąc pod uwagę wady i zalety istniejących rozwiązań, oraz gotowych implementacji (również w bibliotece OpenCV) zdecydowano się na stworzenie własnego algorytmu rozwiązującego problem wydzielania tła z sekwencji obrazów. Brano pod uwagę również podatność algorytmu na zrównoleglanie. Autorskie rozwiązanie, które korzysta z pewnych dobrze znanych rozwiązań zostało opisane w podrozdziale 4.3.

3.3 Interfejsy użytkownika z wykorzystaniem sieci komputerowych

Definicja czym jest sieć komputerowa oraz podstawowe informacje dotyczące sieci nie są poruszane w tej pracy. Informacje przedstawione w tym podrozdziale obejmują jedynie te niezbędne do zrozumienia rozwiązania zaproponowanego w pracy. Dodatkowo zostało przedstawione kilka różnych podejść do rozwiązania problemu komunikacji programu zajmującego się odbieraniem i analizą obrazu z kamery z programem przekładającym wyniki na praktyczne zastosowanie.

Najpierw został przedstawiony problem komunikacji różnych komponentów systemu interfejsu użytkownik-komputer. Następnie omówiona została architektura klient-serwer, zastosowana w pracy. Zostały również przedstawione podstawy dotyczące gniazd sieciowych oraz konsekwencje płynące z wybranego zastosowania. Podczas omówienia treści dotyczących sieci komputerowej zaprezentowane zostało kilka istniejących rozwiązań.

3.3.1 Komunikacja pomiędzy modułami systemu

W systemie wydzielono dwa niezależnie działające programy. Jeden służy do odbierania danych z kamery internetowej, przetwarzaniem ich oraz przekazaniem na wyjście wyni-

ków analizy. Program ten działa w czasie rzeczywistym. Drugi jest wykorzystywany do odbioru i użycia wyników. Uzyskane rezultaty mogą posłużyć zarówno do sterowania myszką komputerową jak również do bardziej zaawansowanych zastosowań. Takimi rozwiązaniami może być zdalne sterowanie innymi urządzeniami, robotami jak również bardziej złożonymi systemami informatycznymi.

Aby informacje z pierwszego programu mogły zostać odebrane przez inny program, musi istnieć interfejs przesyłania informacji. W systemie Windows wyróżniamy wiele możliwości komunikacji, są to na przykład: zdalne wykonywanie procedur (RPC), gniazda internetowe, COM, przetwarzanie potokowe, mapowanie plików i inne.

Komunikacja może odbywać się na różnym poziomie. Inaczej będą mogły komunikować się programy pracujące na jednym komputerze, działające pod obsługą tego samego systemu operacyjnego, a inaczej programy, które do komunikacji wykorzystują sieć. Dodatkowym problemem staje się multiplatformowość, czyli zapewnienie by aplikacje mogły działać na wielu systemach i komputerach, a dane przekazywane pomiędzy nimi były spójne i jednoznaczne.

Poniżej przedstawiono kilka możliwości rozwiązania problemu komunikacji pomiędzy programami. Jako, że w pracy do komunikacji użyto gniazd internetowych, zostały omówione podstawowe informacje dotyczące tej technologii.

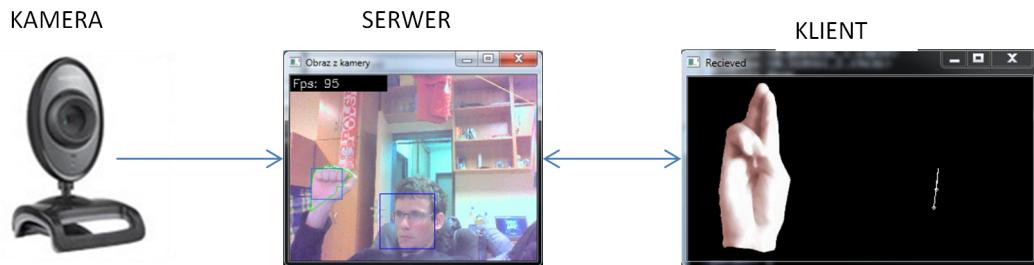
3.3.2 Architektura klient serwer

Najbardziej powszechną metodą na komunikację pomiędzy programami jest wykorzystanie sieci komputerowej. Nie jest to jednak metoda używana często do komunikacji programów znajdujących się na tej samej maszynie, niemniej ma ona i w takich przypadkach swoje zalety. Istnieje wiele wzorców projektowych dotyczących takich rozwiązań, ale najpowszechniej stosowana jest architektura klient-serwer.

Architektura klient-serwer zakłada, że jedna część systemu zapewnia jakieś usługi – jest nią serwer, a druga część używa tych usług. Możliwa jest komunikacja w obie strony. Serwer może udostępniać zarówno informacje, jak i procedury, których wykonanie może odbywać się po jego stronie. Złożoność serwera czy klienta zależy od typu tworzonego systemu. Używając przykładu z tej pracy serwer będzie aplikacją analizującą obraz z kamery i wysyłającą do wszystkich klientów informacje o rozpoznanych gestach i ruchu dloni.

Istnieją różne typy architektury klient serwer, w zależności od ilości komponentów tworzących system. Podział można przedstawić następująco:

- Architektura dwuwarstwowa – przetwarzanie danych i ich składowanie odbywa się w jednym module, który tworzy serwer (przykład na schamacie 3.4).



Schemat 3.4. Przykład architektury dwuwarstwowej na podstawie aplikacji tworzonej z tą pracą.

- Architektura trójwarstwowa – przetwarzanie danych jest oddzielone od przechowywania danych. Najczęściej architekturę tą tworzy baza danych, aplikacja serwera i aplikacja kliencka.
- Architektura wielowarstwowa - przetwarzanie i przechowywanie danych jest rozdzielone na różny sposób. W tej architekturze mogą wystąpić aplikacje pośredniczące, których zadaniem może być jedynie rozdział zadań pomiędzy inne maszyny/aplikacje.

3.3.3 Programowanie gniazd

Implementacja połączeń sieciowych może zostać zrealizowana na wiele sposobów. Jednym z tych sposobów jest implementacja gniazd sieciowych (z angielskiego socket). Jest to implementacja uznawana za najbardziej przenośną pomiędzy systemami operacyjnymi i jest powszechnie stosowana od wielu lat. Pierwszy interfejs gniazd sieciowych (Berkeley sockets) został udostępniony już w 1983 roku wraz z premierą systemu Unix.

Gniazdo jest to pewna abstrakcja, która jest wykorzystywana do wysyłania lub otrzymywania danych z innych procesów. Gniazda używa się do nawiązania połączenia pomiędzy programami, będącymi zarówno na różnych maszynach jak i na tej samej maszynie. Gniazda można również tłumaczyć jako deskryptory plików z danymi, do których można mieć dostęp przez sieć [10].

Istnieją dwa rodzaje gniazd. Są to gniazda strumieniowe i gniazda datagramowe. Te drugie zwane czasem bezpołączeniowymi cechują się odbiorem i wysyłaniem danych bez inicjacyjnego ustanowienia połączenia. Gniazda strumieniowe są to godne zaufania gniazda datagramowe cechujące się dwukierunkowym połączeniem sieciowym.

Gniazda sieciowe jest to interfejs pomiędzy tworzonym programem, a warstwą komunikacyjną w systemie operacyjnym. W tym momencie warto przytoczyć warstwy sieci

i umiejscowić gniazda w tej hierarchii. Główne warstwy sieci (Model OSI) przedstawiają się następująco [10]:

- Warstwa aplikacji.
- Warstwa prezentacji.
- Warstwa sesji.
- Warstwa transportowa.
- Warstwa sieciowa.
- Warstwa łącza danych.
- Warstwa fizyczna.

W tej hierarchii gniazda znajdują się pomiędzy warstwą aplikacji, a warstwą transportową. Oznacza to, że programista może pisać programy nie interesując się tym jak fizycznie dane są przesyłane przez sieć.

W programie zastosowano interfejs gniazd, mimo iż program kliencki i serwer znajdują się na tej samej maszynie, ponieważ rozwiązanie takie jest bardzo łatwo modyfikalne. Umożliwia to podpięcie klientów odbierających informacje z programu znajdującego się na innym komputerze bądź urządzeniu z dostępem do sieci (na przykład sterowanie telewizorem).

3.3.4 Istniejące przykłady rozwiązań

Istnieje wiele programów komunikujących się pomiędzy sobą w systemie. Ich analiza w tej pracy byłaby bezcelowa, jako że każdy z przypadków jest inny, a podejście do niego wymaga osobnych analiz i badań. Niemniej jednak istnieją prace naukowe wyjaśniające podstawowe problemy, z jakimi można spotkać się podczas tworzenia interfejsu do komunikacji. Poniżej przedstawiono taką pracę.

W artykule [24] podjęto temat komunikacji interfejsów użytkownika (GUI) z aplikacją konsolową na platformie Windows. Autorzy prac mieli za zadanie dokonać integracji systemu działającego na main-frame z aplikacją GUI na poziomie komunikacji między procesowej. Praca omawia podstawowe zagadnienia z zakresu tematu takie jak proces, przechowywanie danych na platformach Windows i inne. Zdecydowano się użyć przetwarzania potokowego (z angielskiego pipes) – czyli pewnego rodzaju kolejki gdzie jeden program dostarcza danych, a drugi asynchronicznie odbiera dane z kolejki i je przetwarza.

Integracja z istniejącym systemem wymogła zastosowania aplikacji pośredniczącej (z angielskiego proxy). Pozwoliło to na niezmienianie kodu istniejącego programu zapewniając jednocześnie nową funkcjonalność.

Istnieje bardzo wiele innych rozwiązań, jednak wymienianie i opisywanie ich tutaj jest bezcelowo. Można jedynie nadmienić, że komunikacja pomiędzy procesami lub programami musi być rozwiązana w każdym większym systemie. W tym aspekcie nie można powiedzieć o jedynym słusznym rozwiązaniu, dlatego zawsze warto przeanalizować wady i zalety wszystkich opcji.

3.3.5 Podsumowanie

Biorąc pod uwagę wszystkie możliwości i badając ich wady i zalety w pracy zdecydowano się skorzystać z architektury klient serwer oprogramowanej przy użyciu gniazd WinSOCK. Decyzja ta związana jest z przyszłą możliwością zmiany programu odbierającego dane z analizy ruchu dloni. Architektura ta została uznana jako najbardziej elastyczna i przenośna. Dodatkową jej zaletą jest powszechność i niezawodność, jako że jest ona używana od wielu lat. Więcej informacji na temat sieci w aplikacji można znaleźć w kolejnym rozdziale.

Rozdział 4

Proponowane rozwiązanie systemu

W tym rozdziale została omówiona praca wykonana podczas badań, projektowania i programowania aplikacji rozpoznawania gestów i ruchu dloni. Na początku omówione zostały wymagania i założenia dotyczące pracy, następnie został przedstawiony podział aplikacji na poszczególne moduły. Po wstępnej części kolejno zostały opisane różne aspekty pracy rozpoczynając od procedury zrównoleglenia całej aplikacji. Następnie wydzielony został podrozdział dotyczący segmentacji tła z sekwencji wideo. Na koniec rozdziału znajdują się informacje dotyczące interfejsu zewnętrznego aplikacji i programu klienckiego odbierającego informacje przez sieć, a następnie prezentującego ich na ekranie.

4.1 Wymagania i założenia

Niniejsza praca jest kontynuacją pracy [22] i jej głównym celem jest poprawienie i optymalizacja działania poprzedniej wersji aplikacji. Skupiono się głównie na przyspieszeniu działania programu poprzez wprowadzenie przetwarzania równoległego wszędzie, gdzie jest to możliwe i ma sens. Dodatkowo poprawiono szereg pojedynczych funkcji celem polepszania jakości rozpoznawania.

Głównymi wymaganiami było odciążenie centralnego procesora (CPU) i przerzucenie zapotrzebowania na moc obliczeniową na procesor graficzny. Dodatkowym wymaganiem było by wyniki uzyskiwane po optymalizacji nie były gorsze niż te uzyskiwane wcześniej. Wymaganie to jest o tyle istotne, że najprostszym sposobem optymalizacji mogłoby być zmniejszenie jakości obrazu wejściowego na przykład poprzez zmniejszenie rozdzielczości i przetwarzanie mniejszej ilości danych.

Dodatkowo należało przeprowadzić optymalizację stosowanych metod oraz utworzenie uniwersalnego i przenośnego interfejsu zewnętrznego w celu udostępnienia wyników

do innych zastosowań. Wymaganie dotyczące optymalizacji metod musi uwzględniać algorytm wydzielania statycznego tła, jako że algorytm ten w poprzedniej wersji programu dawał niezadowalające wyniki i był bardzo wrażliwy na zmianę luminescencji otoczenia.

Bardzo ważnym wymaganiem było sprawdzenie wydajności działania programu. Sprawdzenie powinno dotyczyć zarówno całego programu jak również poszczególnych jego części. Szczególną uwagę należało zwrócić na algorytmy implementowane przez autora, czyli między innymi segmentację skóry i wydzielanie tła. Badanie wydajności należało przeprowadzić pod kątem szybkości działania jak również zajętości zasobów komputera. Należało przeprowadzić testy dla działania aplikacji z użyciem procesora graficznego, jak również w trakcie działania jedynie na procesorze głównym.

W pracy skorzystano z biblioteki NVIDIA® CUDA™. Komputer, na którym będzie działała aplikacja musi posiadać kartę graficzną NVIDIA® wspierającą tę technologię. Dodatkowo wymagane jest by w komputerze była zainstalowana kamera.

Program korzysta z interfejsu gniazd sieciowych, dlatego wymagane jest by programy antywirusowe i blokujące ruch sieci w systemie nie blokowały aplikacji. W przypadku nie spełnienia powyższych wymagań aplikacja może nie działać lub działać niepoprawnie.

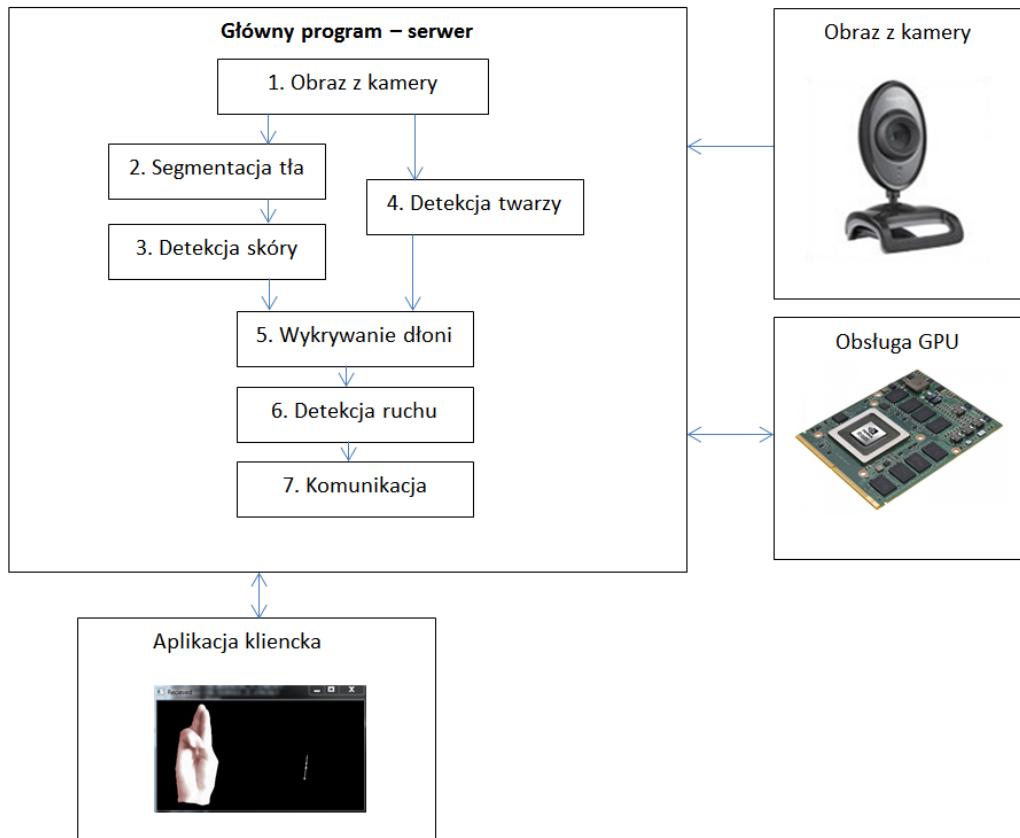
Testy aplikacji wykonano na komputerze z procesorem Intel® CORE™ i7-2720QM CPU @ 2.20Ghz i kartą graficzną NVIDIA® QUADRO 2000M. Procesor główny posiada dodatkową, wbudowaną kartę graficzną Intel® HD Graphics 3000. W komputerze zainstalowane było 8GB pamięci RAM. Testowane kamery podłączone do komputera to: kamera wbudowana w obudowę laptopa 1.3MP oraz zewnętrzna kamera CREATIVE Live! Cam Video IM Pro.

Uruchmienie aplikacji na komputerze stacjonarnym spowoduje uzyskanie znacznie lepszych wyników dla implementacji równoległej od tych zaprezentowanych tutaj. Jest to spowodowane tym, że mobilne procesory graficzne cechują się zdecydowanie gorszą wydajnością aniżeli karty graficzne instalowane w komputerach stacjonarnych.

Logicznie aplikacja została podzielona na dwa główne moduły. Są to: aplikacja serwera – analizująca obraz z kamery oraz aplikacja kliencka – pobierająca wynikowe informacje z aplikacji serwera i wyświetlającą wyniki na ekranie w sposób przystępny użytkownikowi.

Programistycznie aplikacja serwera została dodatkowo podzielona na dwie podaplikacje. Jedna z nich zajmowała się głównym przetwarzaniem, a druga udostępniała funkcje wykonywane na procesorze graficznym z użyciem NVIDIA® CUDA™.

Najbardziej rozbudowaną częścią jest główna aplikacja sterowania przetwarzaniem obrazów. Na tą część składają się poszczególne moduły kolejno przetwarzające obraz z kamery celem uzyskania informacji o dloni znajdującej się na obrazie. Moduły te to, w kolejności przetwarzania:



Schemat 4.1. Schemat systemu rozpoznawania ruchu i gestów dloni.

- Moduł wykrywający twarz – jest wykorzystywany w celu odróżnienia twarzy użytkownika od dłoni.
- Moduł segmentacji statycznego tła – więcej zostało opisane w podrozdziale 4.3.
- Moduł detekcji skóry – z kolorowego obrazu wydzielane są piksele mogące należeć do skóry dłoni ludzkiej.
- Moduł wykrywania dłoni – w tym module następuje dopasowanie obszarów będących potencjalnie dłonią do wzorców treningowych i rozpoznanie gestu dłoni.
- Moduł detekcji ruchu dłoni – rozpoznanie ruchu dłoni na podstawie detekcji ruchu wielu deskryptorów umieszczonych w obszarze dłoni.
- Moduł komunikacji – wysłanie wyników przez sieć do wszystkich znalezionych klientów.

Cały system został przedstawiony na schemacie 4.1.

4.2 Algorytmy przetwarzania równoległego

Jednym z głównych zadań postawionych w tej pracy było zrównoleglenie kodu w celu umożliwienia wykonania go na procesorze graficznym. Podczas prac okazało się, że większość algorytmów wykorzystywanych w programie może zostać zrównoleglona osiągając lepsze wyniki.

Dla niektórych typowych algorytmów w bibliotece OpenCV znajdują się gotowe funkcje, dla których obliczenia wykonywane są na procesorze graficznym. Do takich funkcji należało między innymi wyszukiwanie twarzy na obrazie i algorytm z zakresu sztucznej inteligencji służący do dopasowania wzorca do zbioru treningowego - algorytm siłowy. Dodatkowo w bibliotece OpenCV zostało zaimplementowanych wiele metod pośrednich używanych w systemie. Do takich funkcji należy na przykład:

- Obliczanie przepływu optycznego metodą Lucasa Kanade [4].
- Algorytmy konwersji koloru.
- Algorytmy kopирования obrazów oraz przenoszenia ich z pamięci CPU do pamięci GPU i odwrotnie.
- Operacje logiczne i matematyczne na obrazach (takie jak i, lub, suma, różnica).
- Algorytm progowania binarnego.

Dla wszystkich pozostałych funkcji wymagane było napisanie własnych rozwiązań przez autora w technologii NVIDIA® CUDA™. Były to między innymi:

- Algorytmy morfologiczne – dylatacja, erozja, otwarcie, zamknięcie - w wersji binarnej oraz w wersji w skali szarości (więcej w podrozdziale 4.2.2).
- Algorytmy detekcji skóry – wszystkie dziesięć implementacji musiało zostać zaimplementowane osobno (więcej w podrozdziale 4.2.1).

Dodatkowo należy odnotować, że mimo iż w bibliotece OpenCV znajdowały się podstawowe algorytmy przetwarzania obrazów dla obliczeń na procesorze graficznym niewystarczająca była jedynie ich podmiana. Wymagana była zmiana sposobu przetwarzania całej aplikacji w celu umożliwienia przetwarzania w strumieniu CUDA™. Dodatkowo niektóre algorytmy należało przystosować do wykonywania ich w sposób równoległy, co wiązało się często z nieznaczną zmianą metody rozwiązania.

Warto również zaznaczyć, że bezcelowe byłoby zrównoleglanie wszystkich algorytmów. W niektórych z nich nie znajdował się kod, który mógłby być wykonywany równolegle. Takim algorytmem okazał się algorytm automatycznego dopasowania balansu

bieli w obrazie zaimplementowany przez autora. Algorytm ten bada średni kolor jasnych pikseli na całym obrazie, a następnie dopasowuje balans bieli w celu zapewnienia jak najbardziej zbliżonych kolorów do rzeczywistości. Ma on na celu spowodowanie poprawnego działania algorytmów operujących na kolorach podczas segmentacji skóry. Z racji swojej budowy (obliczania średniej z całego obrazu) równoległa implementacja spowodowałaby jedynie spowolnienie wykonania algorytmu. Ponieważ w procedurze tej nie było miejsc, które można by wykonywać równolegle, nie uzyskano by przyspieszenia działania algorytmu. Jest to zgodne z prawem Amdahla, gdyż jedynie gdy procentowa ilość kodu wykonywanego równolegle jest duża, implementacja na wielu wątkach mogłaby przynieść wymierne korzyści. Został więc on zaimplementowany jedynie dla procesora głównego.

Ponadto należy wziąć pod uwagę, że procesory wielowątkowe są przeważnie wolniej taktowane od procesorów jednowątkowych, dlatego kod sekwencyjny może zostać szybciej wykonany na procesorze CPU. Dodatkowym aspektem jest narzut na komunikację pomiędzy tymi procesorami, co również przekłada się na fakt, że nie każde rozwiązanie da się sensownie zrównoleglić.

Poniżej, w dalszej części tego podrozdziału przedstawiono kolejno algorytmy zrównoleglone przez autora pracy. Są to algorytmy typowe dla rozwiązań zastosowanych tylko w tej pracy, dlatego też niemożliwe było znalezienie gotowego rozwiązania. Ponadto własna implementacja pozwoliła na uzyskanie lepszej wydajności.

4.2.1 Autorski algorytm detekcji skóry

Detekcja skóry (punkt 3 na schemacie 4.1) jest jednym z głównych modułów programu. Tworzy ją dziesięć różnych funkcji wydzielających skórę z obrazu. Wszystkie z tych funkcji są dziełem autora i są opisane w [22]. Tutaj nastąpi jedynie ich krótkie przypomnienie. Jako, że nie są to algorytmy znane i rozpowszechnione, każdy z nich należało zrównoleglić osobno by uzyskać szybsze obliczenia w stosunku do przetwarzania sekwencyjnego oraz odciążenie procesora głównego.

W trakcie tworzenia poprzedniej wersji programu został zaproponowany algorytm detekcji skóry i został on niezmieniony w obecnej implementacji. Jedyną zmianą była implementacja równoległa. Główną koncepcją jest badanie koloru każdego piksela z obrazu wejściowego osobno i determinacja czy piksel ten przedstawia kolor skóry. Algorytm w tej postaci bardzo dobrze ulega dekompozycji gdyż wynik każdego piksela wyjściowego zależy jedynie od wartości jednego piksela mu odpowiadającego. Dzięki temu możliwe jest zrównoleglenie poprzez podzielenie przetwarzania poszczególnych pikseli na kolejne wątki.

Powyższa metoda pozwala na uzyskanie bardzo dobrych wyników, gdyż żadna część

kodu związana z obliczeniami nie musi być synchronizowana, dlatego teoretycznie można osiągnąć bardzo dobrą poprawę wydajności.

Opis poszczególnych algorytmów

Niniejszy podrozdział przypomina budowę poszczególnych algorytmów detekcji skóry.

W celu lepszego rozpoznania czy badany piksel należy do skóry ludzkiej, zaimplementowany został algorytm preselekcji pikseli [21]. Usuwa (zmienia ich kolor na kolor czarny) on piksele nie będące, z dużym prawdopodobieństwem, pikselami należącymi do obszaru skóry ludzkiej. Algorytm ten działa w przestrzeni RGB. Dopiero po wykonaniu preselekcji pikseli następuje rzeczywiste badanie jednym z dokładniejszych algorytmów.

Następnie stworzono pięć funkcji działających w przestrzeni barw RGB. Funkcje te są dziełem autora, a w celu ich identyfikacji nadano im numery. Poniżej krótki opis każdej z funkcji.

- Funkcja 1 w przestrzeni barw RGB – Funkcja opiera się na badaniu zależności poszczególnych składowych. Wartości liczbowe zostały dobrane podczas testów. Funkcja ta jest bardzo prosta, ale i wyniki przez nią osiągane nie są wystarczające do większości zastosowań. Niestety przy jej stosowaniu wiele obszarów niebędących skórą zostaje błędnie zaklasyfikowanych, przez co praktycznie można jej używać tylko w połączeniu z usuwaniem informacji o tle i będąc ubranym w ubrania o kolorze niezblizonym do koloru skóry.
- Funkcja 2 w przestrzeni barw RGB – Funkcja bardziej zaawansowana niż poprzednia opierająca się na złożonych zależnościach kolorów i par kolorów. Wyniki tej funkcji dają bardzo dobre rezultaty w sztucznym oświetleniu. Jednak występuje tu podobny problem jak w przypadku funkcji 1 tzn. błędna klasyfikacja obszarów niebędących skórą
- Funkcja 3 w przestrzeni barw RGB - Prosta funkcja badająca zależności wszystkich składowych koloru. Usunięto niektóre warunki z poprzednich funkcji oraz nieznacznie zmieniono parametry. Jej rezultaty czasem okazują się lepsze od funkcji pierwszej i drugiej.
- Funkcja 4 w przestrzeni barw RGB - Funkcja ta korzysta z zależności poszczególnych składowych do sumy wszystkich składowych. W ten sposób uniknięto wpływu jasności na działanie funkcji.
- Funkcja 5 w przestrzeni barw RGB - Funkcja ta, operuje również na zależnościach pomiędzy poszczególnymi składowymi koloru, daje dobre wyniki tylko w specyficz-

nych warunkach oświetleniowych. W większości przypadków rezultaty przez nią osiągane są niezadowalające.

Dodatkowo zostały napisane funkcje działające w innych przestrzeniach barw:

- Funkcja działająca w przestrzeni barw HSV. Użycie innej przestrzeni barw ma uniezależnić osiągane wyniki od jasności oświetlenia. W celu ulepszenia rezultatów dodano kilka warunków użytych w funkcjach z przestrzeni RGB. Poprawiło to klasyfikację i sprawiło, że funkcja osiąga często najlepsze wyniki w sztucznym oświetleniu.
- Następna funkcja jest wykonywana w przestrzeni barw YCbCr. Podobnie jak poprzednia jest bardziej odporna na różnicę w oświetleniu. Również i w tym algorytmie dobrym rozwiązaniem okazało się dodanie kilku warunków z przestrzeni RGB. Ostatecznie powstała funkcja, która osiąga bardzo dobre wyniki, a w niektórych warunkach akwizycji daje najlepsze rezultaty ze wszystkich zaimplementowanych algorytmów.

W trakcie badań stwierdzono, że nie zawsze algorytm preselekcji pikseli daje pożądane wyniki. Z powyższych algorytmów stworzono wynikowych dziesięć procedur zgodnie tabelą 4.1. Dlatego ostatecznie stworzono implementację złożoną z kombinacji algorytmów z algorytmem preselekcji pikseli.

Wyniki badań

W celu przetestowania implementacji funkcji należało zebrać czasy wykonania każdej z poszczególnych procedur dla różnych rozdzielczości. Przebadano czas wykonywania funkcji zarówno na procesorze głównym jak również na procesorze graficznym. W tym celu obliczano liczbę cykli procesora pomiędzy rozpoczęciem, a zakończeniem wykonywania przetwarzania, a następnie wynik zamieniano na czas w sekundach (w zależności od szybkości taktowania procesora).

W tabelach 4.2, 4.3 i na wykresach 4.2 - 4.5 przedstawiono wyniki badań czasów wykonania funkcji rozpoznawania koloru skóry dla różnych rozdzielczości.

Wyniki zostały zebrane podczas normalnego działania aplikacji na nieobciążonym komputerze. Wszystkie czasy podane są w sekundach. Dokładność wyniku jest uzyskana dzięki zliczaniu ilości cykli procesora pomiędzy rozpoczęciem, a zakończeniem wykonywania algorytmu wiele razy, a następnie braniami wartości średniej z rezultatów.

Badaniu został poddany jedynie kod odpowiedzialny za wykonanie segmentacji koloru skóry. Pominięto operacje przenoszenia danych z i na procesor graficzny. Należy wziąć pod uwagę, że taki transfer danych występuje jedynie raz podczas całego przetwarzania

| Nazwa funkcji wynikowej | Użyty algorytm | Czy używa algorytmu preselekcji |
|-------------------------|-----------------------------------|---------------------------------|
| HSV+ | Algorytm w przestrzeni barw HSV | Tak |
| YCbCr+ | Algorytm w przestrzeni barw YCbCr | Tak |
| RGB1 | Funkcja 1 w przestrzeni barw RGB | Tak |
| RGB2 | Funkcja 2 w przestrzeni barw RGB | Tak |
| RGB3 | Funkcja 3 w przestrzeni barw RGB | Tak |
| RGB4 | Funkcja 4 w przestrzeni barw RGB | Tak |
| RGB5 | Funkcja 5 w przestrzeni barw RGB | Tak |
| HSV- | Algorytm w przestrzeni barw HSV | Nie |
| YCbCr- | Algorytm w przestrzeni barw YCbCr | Nie |
| Algorytm preselekcji | Brak | Tak |

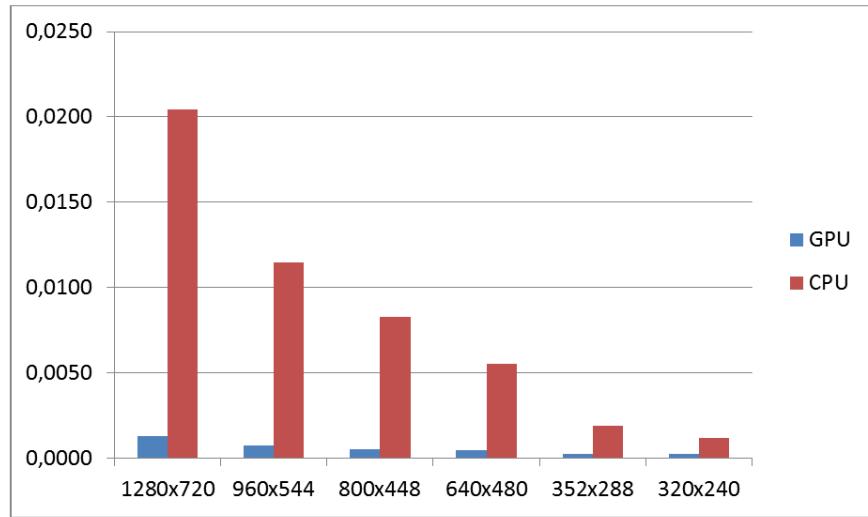
Tabela 4.1. Nazwy poszczególnych funkcji rozpoznawania skóry.

| Nazwa funkcji | Wykonywana na procesorze | 1280 na 720 | 960 na 544 | 800 na 448 | 640 na 480 | 352 na 288 | 320 na 240 |
|----------------------|--------------------------|-------------|------------|------------|------------|------------|------------|
| HSV+ | GPU | 0,0015 | 0,0008 | 0,0006 | 0,0006 | 0,0003 | 0,0003 |
| | CPU | 0,0274 | 0,0194 | 0,0114 | 0,0052 | 0,0015 | 0,0013 |
| YCbCr+ | GPU | 0,0015 | 0,0008 | 0,0006 | 0,0005 | 0,0003 | 0,0003 |
| | CPU | 0,0230 | 0,0169 | 0,0108 | 0,0062 | 0,0013 | 0,0012 |
| RGB1 | GPU | 0,0013 | 0,0008 | 0,0005 | 0,0005 | 0,0003 | 0,0002 |
| | CPU | 0,0186 | 0,0133 | 0,0081 | 0,0051 | 0,0014 | 0,0011 |
| RGB2 | GPU | 0,0014 | 0,0008 | 0,0006 | 0,0005 | 0,0003 | 0,0002 |
| | CPU | 0,0220 | 0,0163 | 0,0106 | 0,0064 | 0,0017 | 0,0012 |
| RGB3 | GPU | 0,0013 | 0,0007 | 0,0006 | 0,0005 | 0,0003 | 0,0002 |
| | CPU | 0,0193 | 0,0084 | 0,0078 | 0,0055 | 0,0015 | 0,0011 |
| RGB4 | GPU | 0,0013 | 0,0007 | 0,0006 | 0,0005 | 0,0003 | 0,0002 |
| | CPU | 0,0192 | 0,0081 | 0,0072 | 0,0055 | 0,0015 | 0,0011 |
| RGB5 | GPU | 0,0013 | 0,0008 | 0,0006 | 0,0005 | 0,0003 | 0,0002 |
| | CPU | 0,0182 | 0,0082 | 0,0068 | 0,0055 | 0,0015 | 0,0011 |
| HSV- | GPU | 0,0009 | 0,0005 | 0,0004 | 0,0003 | 0,0002 | 0,0002 |
| | CPU | 0,0231 | 0,0090 | 0,0081 | 0,0065 | 0,0017 | 0,0013 |
| YCbCr- | GPU | 0,0008 | 0,0005 | 0,0003 | 0,0003 | 0,0002 | 0,0001 |
| | CPU | 0,0189 | 0,0079 | 0,0066 | 0,0054 | 0,0015 | 0,0011 |
| Algorytm preselekcji | GPU | 0,0015 | 0,0009 | 0,0008 | 0,0007 | 0,0005 | 0,0005 |
| | CPU | 0,0148 | 0,0069 | 0,0057 | 0,0040 | 0,0054 | 0,0013 |

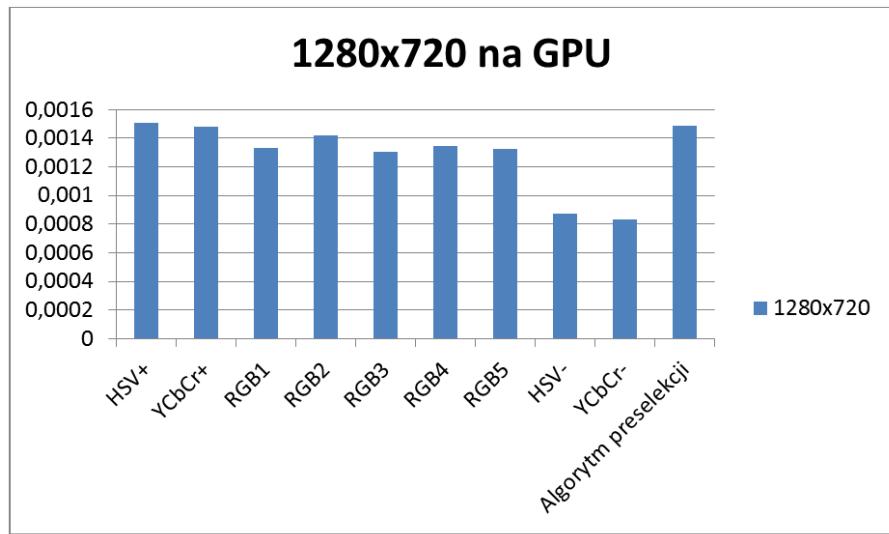
Tabela 4.2. Uśrednione czasy wykonania funkcji dla różnych rozdzielczości.

| Rozdzielcość | Obliczenia na GPU | Obliczenia na CPU | Przyspieszenie |
|--------------|-------------------|-------------------|----------------|
| 1280 na 720 | 0,0013 | 0,0205 | 15,849 |
| 960 na 544 | 0,0007 | 0,0115 | 15,532 |
| 800 na 448 | 0,0006 | 0,0083 | 14,893 |
| 640 na 480 | 0,0005 | 0,0055 | 11,126 |
| 352 na 288 | 0,0003 | 0,0019 | 6,699 |
| 320 na 240 | 0,0002 | 0,0012 | 4,848 |

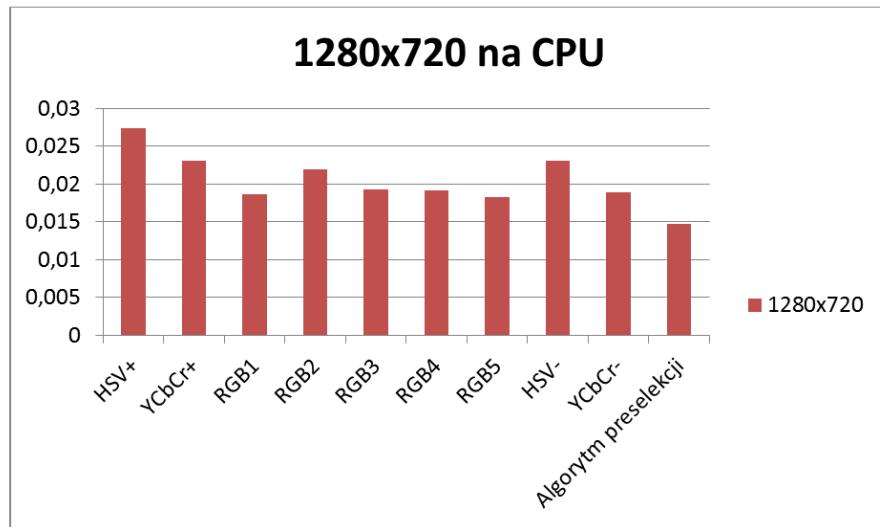
Tabela 4.3. Uśrednione czasy wykonania wszystkich funkcji dla różnych rozdzielczości.



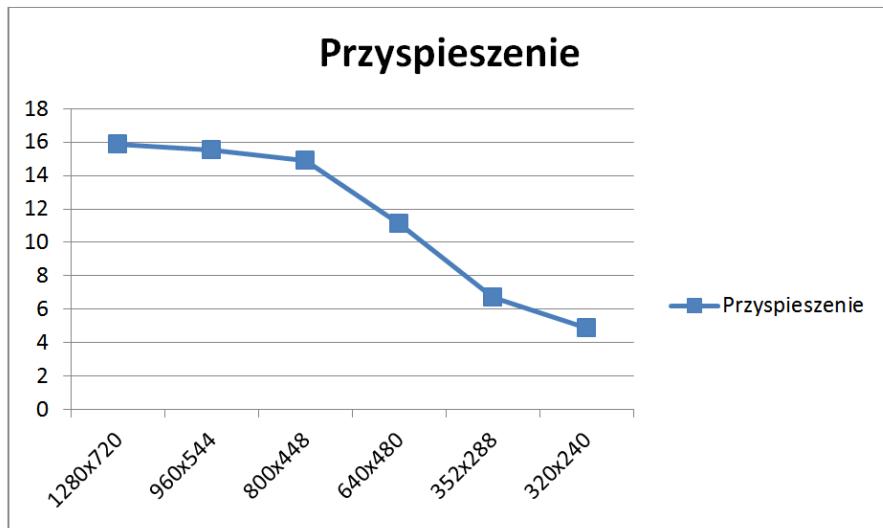
Wykres 4.2. Uśrednione czasy wykonywania algorytmu rozpoznawania skóry.



Wykres 4.3. Uśrednione czasy wykonywania poszczególnych funkcji dla procesora graficznego.



Wykres 4.4. Uśrednione czasy wykonania poszczególnych funkcji dla procesora CPU.



Wykres 4.5. Przyspieszenie wykonywania algorytmów uzyskane dzięki użyciu obliczeń wykonywanych na procesorze graficznym CPU.

jednego obrazu pobranego z kamery. Dlatego też czas przenoszenia danych nie jest istotny dla zbadania jakości rozwiązania.

Każdy wynik w tabelach dotyczących pojedynczych funkcji i jest wartością średnią ze stu pomiarów. Ogólnie w celu zbadania przyspieszenia działania segmentacji koloru skóry wykonano sześć tysięcy pomiarów czasu wykonywania algorytmów.

Wnioski

Powyższe wyniki pokazują, że już nawet przy najniższej badanej rozdzielczości korzyść z użycia procesora graficznego do obliczeń jest bardzo wysoki. Osiągnięto od pięciokrotnego aż do piętnastokrotnego przyspieszenia wykonywania algorytmu. Wynik ten okazał się zdecydowanie lepszy od oczekiwanej.

Przy niewielkich rozdzielczościach nakład na komunikację i synchronizację wątków jest zdecydowanie większy, dlatego też prawdziwe przyspieszenie działania uzyskuje się dopiero przy wysokich rozdzielczościach.

Zgodnie z oczekiwaniami przy większych rozdzielczościach obserwuje się większą skalowalność dla rozwiązania na procesorze graficznym aniżeli na procesorze głównym.

Analizując wyniki poszczególnych funkcji można zauważyc, że czas wykonania funkcji HSV- i YCbCr- jest wyraźnie krótszy od pozostałych dla wykonania na procesorze graficznym. Ma to związek z różnicą implementacyjną tych funkcji. Wszystkie pozostałe funkcje składają się z dwóch składowych funkcji wykonywanych jedna po drugiej. Są to funkcja filtru wstępnego i właściwa funkcja. Dla funkcji HSV- i YCbCr- filtr wstępny jest pomijany, dlatego też obliczenia w tym przypadku są szybsze. Więcej na temat algorytmów użytych podczas segmentacji skóry można przeczytać w pracy inżynierskiej [22] jak również na początku podrozdziału 4.2.1.

Wyniki rozpoznawania osiągane przez zrównoleglone algorytmy są identyczne jak te uzyskiwane przez algorytmy wykonywane sekwencyjnie. Jest to spowodowane faktem, że dokonywane są dokładnie te same obliczenia, a kolejność ich wykonywania nie ma wpływu na wynik końcowy.

4.2.2 Zrównoleglone algorytmy morfologii

W trakcie pisania pracy analizowano, które funkcje wykonują się najdłużej, a ich optymalizacja jest kluczowa. Okazało się, że najbardziej spowalniającymi pracę całej aplikacji algorytmami są operacje morfologiczne. Dlatego też zdecydowano się poświęcić więcej czasu w celu ich jak najlepszej optymalizacji.

Pierwszym podejściem było użycie algorytmów morfologii zaimplementowanych w bibliotece OpenCV dla procesora graficznego. Okazało się jednak, że implementacje te

odznaczały się bardzo słabą wydajnością. Są one bowiem napisane tak, by były jak najbardziej elastyczne, co w praktyce oznacza, że dla obecnego przypadku są zbyt rozbudowane i przez to wolne.

W większości przypadków w pracy używana była morfologia binarna, dla której możliwe były daleko idące optymalizacje opisane w kolejnych podrozdziałach.

Wymagania, cel i założenia

Celem jest stworzenie algorytmu morfologii w sposób by jego wykonywanie odbywało się w możliwie jak najkrótszym czasie. W pracy występuje duża liczba morfologii binarnej stąd głównie ona będzie brana pod uwagę przy optymalizacji.

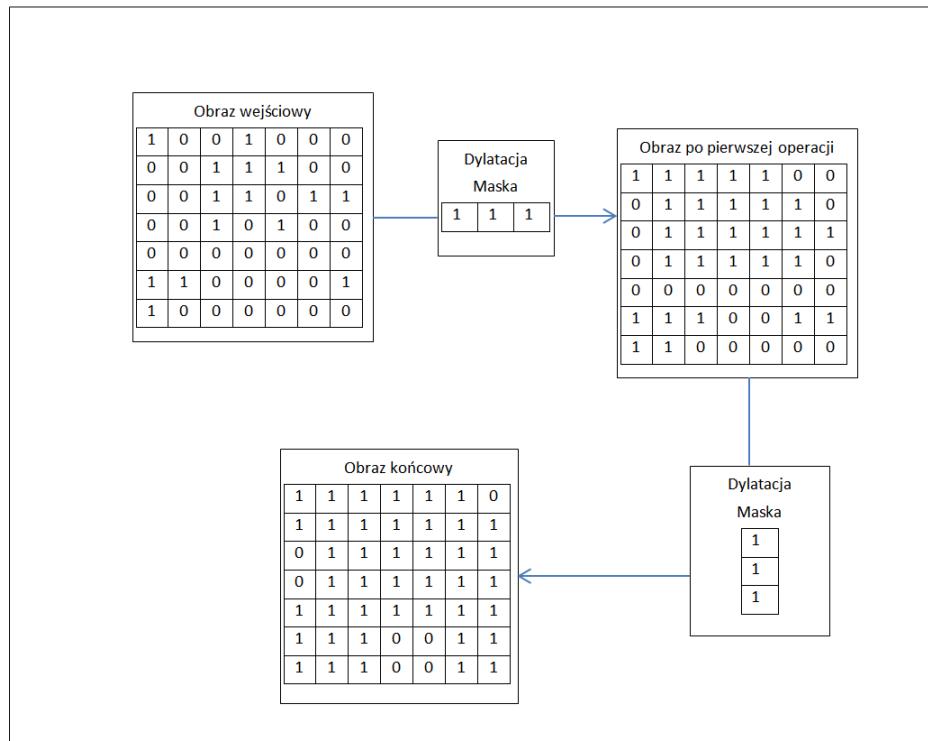
W pracy założono, że korzysta się z kwadratowych elementów strukturujących o jednakowych wartościach. W tym miejscu warto zaznaczyć, że w pracy testowano również możliwość użycia innych masek jak na przykład okrągłych. Dawały one nieco lepsze rezultaty jednak w stosunku do różnicicy w czasach wykonywania zdecydowano się użyć wydajniejszych masek kwadratowych.

Algorytmy morfologii binarnej

Dokładny opis algorytmu znajduje się w podrozdziale 3.2.1. Dla przypomnienia wartość piksela wyjściowego jest równa wartości maksymalnej (dylatacja) bądź minimalnej (erozja) znajdującej się w masce. Implementacja algorytmów morfologicznych w OpenCV daje bardzo dobre wyniki, ale jedynie dla wykonywania na procesorze CPU. Implementacja GPU jest implementacją naiwną o złożoności kwadratowej, dlatego by zachować ciągłość przetwarzania na procesorze graficznym należało dokonać optymalizacji algorytmu dla procesora graficznego.

Badania nad algorytmem doprowadziły do wniosku, że dla danych założeń (kwadratowa maska) możliwa jest dekompozycja problemu na dwa pomniejsze problemy. Zamiast obliczania morfologii dla kwadratowej maski możliwe jest dwukrotne obliczenie wartości wynikowej dla liniowego elementu strukturującego. Wpierw obliczany jest wynik dla poziomej linii, a następnie wynik ten jest poddany morfologii z maską pionową. Warto zaznaczyć, że nie jest istotna kolejność wykonywania tych procedur, dlatego równie poprawną jest metoda gdzie najpierw obliczany jest wynik dla pionowej maski, a następnie poziomej.

Powyższe rozwiązanie pozwala na zmniejszenie złożoności obliczeniowej algorytmu morfologii z $O(n \cdot m^2)$ gdzie n to wielkość obrazu, a m oznacza wielkość maski do $O(n \cdot 2 \cdot m)$, czyli w ogólnym zapisie $O(n \cdot m)$.



Schemat 4.6. Zdekomponowana operacja dylatacji.

Przykładowe obliczenia dla algorytmu zostały przedstawione na schemacie 4.6. Przedstawiona została operacja dylatacji dla maski o wielkości 3.

Późniejsze analizy rozwiązań, które znajduje się w bibliotece OpenCV doprowadziły do wniosku, że dla takiej maski również i w implementacji dla CPU występuje dekompozycja podobna jak ta zaproponowana powyżej.

Dla zbadania poprawności rozwiązania zostały wykonane testy na rzeczywistych danych. Rozwiązanie okazało się poprawne i dużo szybsze od rozwiązania bez dekompozycji. Wykonano szereg testów wydajnościowych opisanych w kolejnym podrozdziale.

Badania i wyniki

W pracy przebadano czas wykonywania operacji otwarcia (erozja, a następnie dylatacja) dla czterech implementacji algorytmu. Są to kolejno metoda zaimplementowana przez autora z dekompozycją problemu na procesorze GPU, metoda zaimplementowana przez autora bez dekompozycji, algorytm zaimplementowany w bibliotece OpenCV dla procesora centralnego oraz dla procesora graficznego.

Dla każdego z tych algorytmów przetestowano szybkość wykonywania dla kwadratowej maski. Testy przeprowadzona dla następującej ilości iteracji algorytmu: 1, 5, 10, 15, 20, 25, 30. Dla każdego testu o zadanej ilości iteracji wykonano sto pomiarów. Wszystkie

| 320 na 240 | 1 | 5 | 10 | 15 | 20 | 25 | 30 |
|------------------|--------|--------|--------|--------|--------|--------|--------|
| Z dekompozycja | 0,0002 | 0,0005 | 0,0007 | 0,0010 | 0,0012 | 0,0014 | 0,0016 |
| Bez dekompozycji | 0,0005 | 0,0051 | 0,0181 | 0,0391 | 0,0674 | 0,1034 | 0,1465 |
| OpenCV CPU | 0,0001 | 0,0002 | 0,0003 | 0,0005 | 0,0006 | 0,0007 | 0,0008 |
| OpenCV GPU | 0,0010 | 0,0070 | 0,0201 | 0,0467 | 0,0842 | 0,1111 | 0,1332 |

Tabela 4.4. Uśrednione czasy wykonania morfologii dla rozdzielczości 320 na 240.

| 352 na 288 | 1 | 5 | 10 | 15 | 20 | 25 | 30 |
|------------------|--------|--------|--------|--------|--------|--------|--------|
| Z dekompozycja | 0,0004 | 0,0006 | 0,0009 | 0,0012 | 0,0015 | 0,0017 | 0,0020 |
| Bez dekompozycji | 0,0006 | 0,0067 | 0,0238 | 0,0512 | 0,0885 | 0,1358 | 0,1927 |
| OpenCV CPU | 0,0001 | 0,0002 | 0,0004 | 0,0005 | 0,0008 | 0,0009 | 0,0010 |
| OpenCV GPU | 0,0012 | 0,0085 | 0,0245 | 0,0576 | 0,1046 | 0,1390 | 0,1665 |

Tabela 4.5. Uśrednione czasy wykonania morfologii dla rozdzielczości 352 na 288.

badania zostały przeprowadzone dla rozdzielczości: 320 na 240, 352 na 288, 640 na 480, 800 na 448, 960 na 544 oraz 1280 na 720. Łącznie daje to czternaście tysięcy czterysta pomiarów. Uśrednione dane zostały przedstawione w tabelach 4.4 - 4.9.

Wyniki osiągane przez poszczególne algorytmy morfologii były zawsze identyczne. Wyniki różniłyby się jedynie w przypadku gdyby została użyta maska inna niż kwadratowa. Niemożliwe wtedy byłaby zastosowanie dekompozycji, tak więc przypadek ten nie był rozpatrywany.

Wnioski

Optymalizacja implementacji algorytmów morfologii przyniosła korzyść dla kwadratowej maski. Możliwa dekompozycja problemu została zaimplementowana dla takiego elementu strukturyjącego i daje widoczne przyspieszenie wykonywania algorytmu.

Ważnym wnioskiem płynącym z tych rozważań jest fakt, że zrównoleglanie nie jest

| 640 na 480 | 1 | 5 | 10 | 15 | 20 | 25 | 30 |
|------------------|--------|--------|--------|--------|--------|--------|--------|
| Z dekompozycja | 0,0007 | 0,0015 | 0,0024 | 0,0032 | 0,0041 | 0,0050 | 0,0059 |
| Bez dekompozycji | 0,0017 | 0,0198 | 0,0712 | 0,1540 | 0,2671 | 0,4111 | 0,5860 |
| OpenCV CPU | 0,0003 | 0,0006 | 0,0011 | 0,0015 | 0,0020 | 0,0025 | 0,0027 |
| OpenCV GPU | 0,0021 | 0,0200 | 0,0646 | 0,1443 | 0,2573 | 0,3637 | 0,4635 |

Tabela 4.6. Uśrednione czasy wykonania morfologii dla rozdzielczości 640 na 480.

| 800 na 448 | 1 | 5 | 10 | 15 | 20 | 25 | 30 |
|------------------|--------|--------|---------|--------|--------|--------|--------|
| Z dekompozycja | 0,0008 | 0,0017 | 0,0027 | 0,0038 | 0,0049 | 0,0059 | 0,0068 |
| Bez dekompozycji | 0,0020 | 0,0231 | 10,0000 | 0,1794 | 0,3113 | 0,4793 | 0,6829 |
| OpenCV CPU | 0,0003 | 0,0006 | 0,0012 | 0,0017 | 0,0024 | 0,0027 | 0,0032 |
| OpenCV GPU | 0,0025 | 0,0240 | 0,0772 | 0,1720 | 0,3054 | 0,4358 | 0,5316 |

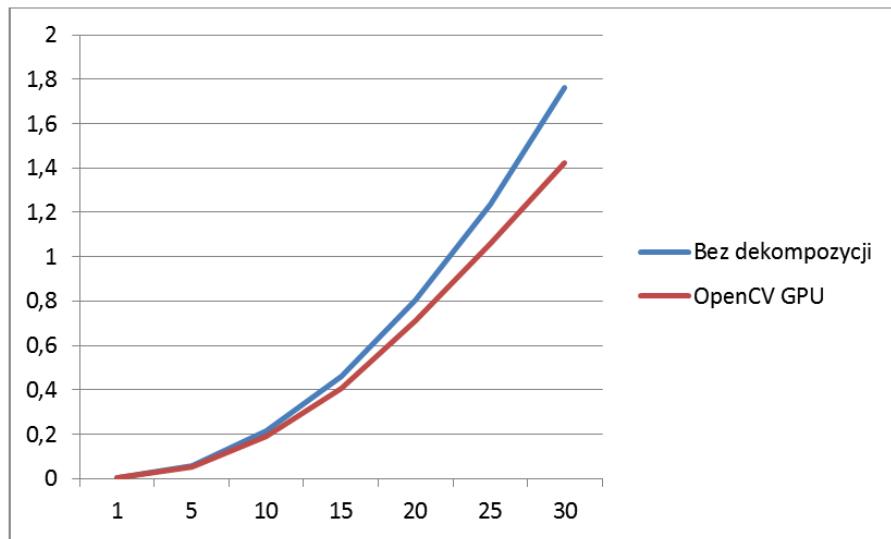
Tabela 4.7. Uśrednione czasy wykonania morfologii dla rozdzielczości 800 na 448.

| 960 na 544 | 1 | 5 | 10 | 15 | 20 | 25 | 30 |
|------------------|--------|--------|--------|--------|--------|--------|--------|
| Z dekompozycja | 0,0012 | 0,0024 | 0,0039 | 0,0054 | 0,0069 | 0,0084 | 0,0099 |
| Bez dekompozycji | 0,0029 | 0,0336 | 0,1206 | 0,2612 | 0,4540 | 0,6994 | 0,9970 |
| OpenCV CPU | 0,0005 | 0,0009 | 0,0016 | 0,0024 | 0,0033 | 0,0040 | 0,0045 |
| OpenCV GPU | 0,0033 | 0,0329 | 0,1090 | 0,2406 | 0,4223 | 0,6124 | 0,8152 |

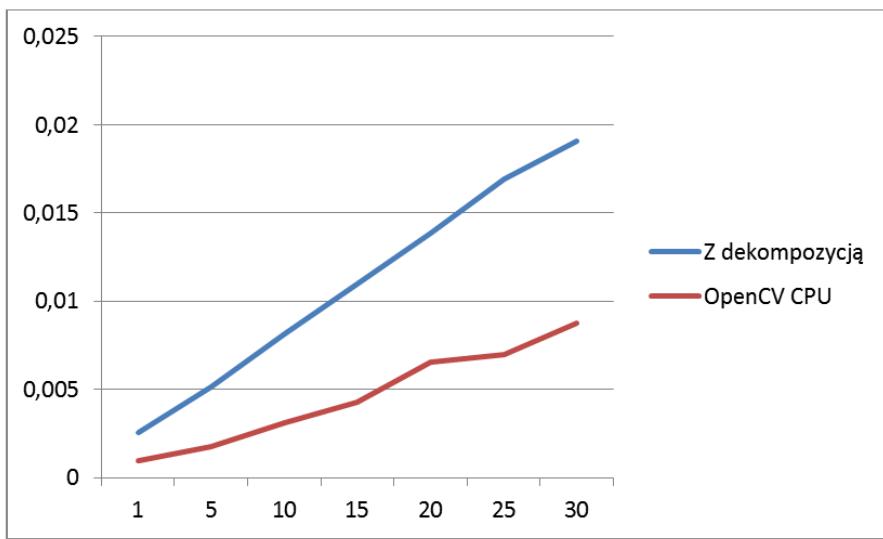
Tabela 4.8. Uśrednione czasy wykonania morfologii dla rozdzielczości 960 na 544.

| 1280 na 720 | 1 | 5 | 10 | 15 | 20 | 25 | 30 |
|------------------|--------|--------|--------|--------|--------|--------|--------|
| Z dekompozycją | 0,0025 | 0,0051 | 0,0081 | 0,0110 | 0,0138 | 0,0169 | 0,0191 |
| Bez dekompozycji | 0,0051 | 0,0596 | 0,2131 | 0,4619 | 0,8026 | 1,2370 | 1,7639 |
| OpenCV CPU | 0,0010 | 0,0018 | 0,0031 | 0,0042 | 0,0065 | 0,0070 | 0,0088 |
| OpenCV GPU | 0,0053 | 0,0551 | 0,1896 | 0,4065 | 0,7115 | 1,0579 | 1,4249 |

Tabela 4.9. Uśrednione czasy wykonania morfologii dla rozdzielczości 1280 na 720.



Wykres 4.7. Uśrednione czasy wykonania algorytmu autora bez dekompozycji na procesorze GPU oraz implementacji OpenCV dla procesora GPU dla rozdzielczości 1280 na 720.



Wykres 4.8. Uśrednione czasy wykonania algorytmu autora bez dekompozycji na procesorze GPU oraz implementacji OpenCV dla procesora GPU dla rozdzielczości 1280 na 720.

jedyną możliwością przyspieszenia działania algorytmu, a czasem istnieją inne, dużo lepsze rozwiązania. Najważniejszym elementem, który należy poddać optymalizacji jest sam algorytm. Jeżeli możliwe jest zmniejszenie złożoności obliczeniowej algorytmu to w zdecydowanej większości przypadków taki zabieg da zdecydowanie lepsze efekty niż zrównoleglenie nieoptymalnego rozwiązania. Dopiero w przypadku, gdy algorytm jest doprowadzony do możliwie najlepszej złożoności obliczeniowej, w ramach dalszej optymalizacji warto przeanalizować zrównoleglenie rozwiązania.

Z powyższych wyników widać również, że implementacja dla procesora centralnego w bibliotece OpenCV jest zadziwiająco dobra i daje lepsze wyniki nawet od implementacji na procesorze graficznym. Wniosek z tego płynący jest taki, że operacja morfologii operuje na dużej ilości danych z całego obrazu, dlatego też narzuty pamięciowe są na tyle istotne, że w przypadku gdy jest to jedyna operacja na obrazie, którą należy wykonać nie jest opłacalne używanie do tego procesora graficznego.

4.2.3 Zrównoleglanie programu jako całości

Po zrównolegleniu poszczególnych metod należy przeprowadzić zrównoleglenie programu jako całości. Zrównoleglanie pojedynczych metod dla przetwarzania obrazów nie ma sensu, ponieważ gdy każdorazowo miałby następować transfer danych pomiędzy kartą graficzną, a procesorem głównym nie przyniosłoby to wymiernych korzyści.

Dopiero, gdy całe przetwarzanie przeniesie się na procesor graficzny, a transfer da-

nych występuje jednokrotnie można mówić o zrównolegleniu całego programu. W pracy algorytm rozpoznawania gestów i ruchu dloni podzielono na poszczególne składowe. Nie wszystkie z nich zostały zrównoleglone, ponieważ część z nich operowała na danych z całego obrazu, przez co zrównoleglanie nie dawało żadnych korzyści. Dodatkowo warto zaznaczyć, że operacje które nie zostały przeniesione do wykonania na procesor graficzny wykonują się na tyle szybko, że ich wpływ jest pomijalny w stosunku do przetwarzania jako całości.

Wymagania cel i założenia

Głównym wymaganiem zrównoleglenia aplikacji jako całości było uzyskanie krótszego czasu wykonania programu jako całości, lub umożliwienie na wykonanie bardziej zaawansowanych algorytmów nie tracąc na szybkości działania aplikacji.

Dodatkowo niezbędne było uzyskanie przynajmniej porównywalnych bądź lepszych wyników rozpoznawania gestów oraz detekcji ruchu w rzeczywistym środowisku, tj. w domu, w pracy.

Do poprawnego działania programu nie jest wymagany dodatkowy sprzęt typu ręka-wiczki, znaczniki, bądź wyspecjalizowane sensory na przykład Kinect firmy Microsoft®.

Zastosowane rozwiązanie

Cały system został przedstawiony na schemacie 4.1. Odwołując się do poszczególnych elementów systemu poniżej przedstawiono operacje wykonywane na procesorze graficznym. Są to:

- Segmentacja tła.
- Wykrywanie twarzy.
- Segmentacja koloru skóry.
- Dopasowanie gestu do wykrytego obszaru dloni (w końcowej wersji systemu jednak nie, porównaj z wnioskami w tym podrozdziale).
- Śledzenie ruchu dloni.

Pewne operacje dotyczące komunikacji, automatycznego balansu bieli, rozpoznanie gestu, uaktualniania deskryptorów służących do śledzenia dloni i kilka innych są wykonywane na procesorze głównym.

W pracy zastosowano przetwarzanie potokowe używając strumieni CUDA™. Są to operacje wykonywane na procesorze graficznym asynchronicznie zgodnie z kolejnością zgłoszenia. Więcej na temat przetwarzania strumieniowego w NVIDIA® CUDA™

można przeczytać w prezentacji Steve Rennich [17]. Przetwarzanie w jednym strumieniu jest możliwe dla segmentacji tła, wykrywania twarzy oraz segmentacji koloru skóry. Dla wszystkich tych operacji przeniesienie danych z pamięci głównej do pamięci procesora graficznego oraz w przeciwną stronę występuje tylko raz. Dodatkowo zgłoszenie do wykonania jakiejś procedury po stronie procesora graficznego (o ile nie jest to procedura wymagająca transferu danych, na przykład operacja ściągnięcia danych z karty graficznej) następuje asynchronicznie i nie blokuje wykonywania kodu dla procesora głównego.

Dzięki takiemu rozwiązaniu nie tylko uzyskuje się większą wydajność, ale również pozwala się odciążyć główny procesor od obliczeń. Skutkuje to mniejszą zajętością zasobów wymaganych do poprawnej pracy systemu operacyjnego. Dodatkowo warto zaznaczyć, że obecnie w wielu komputerach instalowane są dwie karty graficzne. W przypadku tej aplikacji daje to minimalne obciążenie zasobów potrzebnych do pracy systemu operacyjnego, co jest o tyle istotne, że aplikacja ta ma służyć jako interfejs komputer-użytkownik.

Badania i wyniki

W tabelach i na wykresach poniżej przedstawiono wyniki badań szybkości działania programu oraz obciążenia procesora i karty graficznej podczas pracy systemu. Wyniki w tabeli 4.10 są częściowe i mają na celu pokazanie specyficznych czasów w różnych etapach działania programu. Przedziały zostały dobrane tak, że czasy wykonania programu w poszczególnych przedziałach były bardzo zbliżone do siebie i nie było potrzeby prezentowania wszystkich danych. Różnice wynikają z architektury programu i są dokładniej opisane we wnioskach.

Pozostałe wyniki są wynikami średnimi uzyskanymi każdorazowo z przynajmniej stu badań (w większości dwustu). Badania dotyczące wykorzystywania zasobów systemowych zostały wykonane za pomocą zewnętrznych programów. Warto zaznaczyć, że badania zostały wykonane na procesorze głównym o ośmiu wątkach logicznych, a wykorzystanie procesora tyczy się wszystkich ośmiu wątków. Dlatego też wykorzystanie procesora na poziomie 12,5% jest równoznaczne z wykorzystaniem procesora na poziomie 100% na maszynie jednoprocesorowej. Wykorzystanie procesora powyżej 12,5% jest równoznaczne z wykorzystaniem przez program przynajmniej dwóch wątków (co mam miejsce w implementacji dla CPU ponieważ rozpoznawanie twarzy odbywa się w osobnym wątku).

Program był testowany jako jedyny program obciążający system. Wyniki z użycia procesora są wynikami ogólnymi zajętości procesora. Badano jednak jaki czas obliczeń procesora został przydzielony do wykonywania programu rozpoznawania gestów i wynik był bardzo bliski 95%. Można więc założyć, że podane wyniki dotyczą tylko badanego

| Numer klatki | CPU - przed rozpoznaniem gestu | CPU - rozpoznanie gestu + aktualizacja obszaru dłoni | GPU - przed rozpoznaniem gestu | GPU - rozpoznanie gestu + aktualizacja obszaru dłoni | CPU rozpoznanie gestu + GPU aktualizacja obszaru dłoni |
|-------------------------|--------------------------------|--|--------------------------------|--|--|
| 1 | 0,0054 | 0,0002 | 0,0424 | 0,0134 | 0,0002 |
| Średnia z 2-36 | 0,0041 | 0,0001 | 0,0051 | 0,0133 | 0,0002 |
| 37 | 0,0061 | 0,0001 | 0,0072 | 0,0134 | 0,0001 |
| Średnia z 38-56 | 0,0061 | 0,0001 | 0,0069 | 0,0134 | 0,0002 |
| 57 | 0,0082 | 0,0002 | 0,0114 | 0,0131 | 0,0001 |
| Średnia z 58 i dalszych | 0,0092 | 0,0001 | 0,0117 | 0,0101 | 0,0002 |

Tabela 4.10. Uśrednione czasy wykonywania poszczególnych części programu w rozdzielcości 320 na 240.

programu.

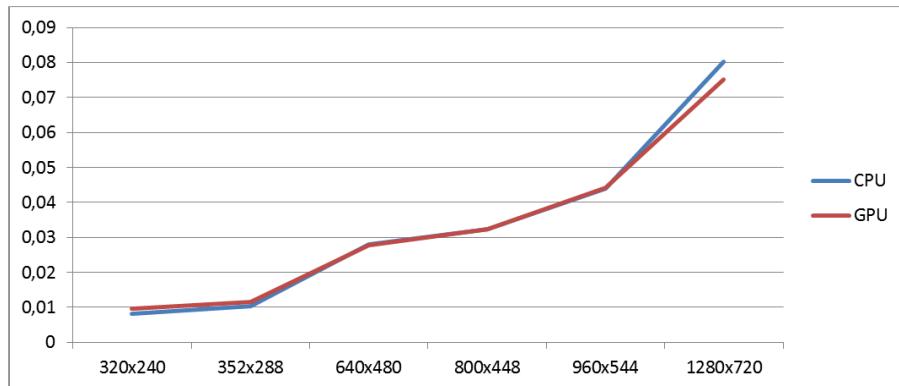
Wnioski

Wyniki pokazują, że udało się spełnić założenia. Aplikacja z użyciem procesora graficznego działa porównywalnie szybko z aplikacją działającą jedynie na procesorze głównym. Ponadto wyniki wykorzystania zasobów systemowych podczas działania aplikacji pokazują, że system jest mniej obciążony w przypadku działania aplikacji, gdy używa ona procesora graficznego. Poniżej dokładnie omówiono wszystkie wyniki oraz wytłumaczono uzyskane rezultaty.

Tabela 4.10 pokazuje wyniki jakie osiągała aplikacja podczas działania w docelowej rozdzielcości 320 na 240 pikseli. Dla tej rozdzielcości implementacja na procesorze graficznym uzyskuje nieznacznie gorsze wyniki czasowe od implementacji na samym procesorze głównym. Warto zauważyć pewne skoki czasowe w 37 i 57 klatce działania programu. Pierwsza z nich następuje, gdy algorytm usuwania tła skumuluje już odpowiednią liczbę ramek tła i od tego momentu następuje segmentacja tła. Powoduje to nieznaczne spowolnienie działania aplikacji, zarówno implementacji hybrydowej jak i implementacji

| Rozdzielczość | CPU - przed rozpoznaniem gestu | CPU - rozpoznanie gestu + aktualizacja obszaru dłoni | GPU - przed rozpoznaniem gestu | GPU - rozpoznanie gestu + aktualizacja obszaru dłoni | CPU rozpoznanie gestu + GPU aktualizacja obszaru dłoni |
|---------------|--------------------------------|--|--------------------------------|--|--|
| 320 na 240 | 0,00814 | 0,00013 | 0,00960 | 0,01315 | 0,00016 |
| 352 na 288 | 0,01021 | 0,00013 | 0,01160 | 0,01121 | 0,00015 |
| 640 na 480 | 0,02804 | 0,00013 | 0,02773 | 0,01252 | 0,00015 |
| 800 na 448 | 0,03231 | 0,00014 | 0,03233 | 0,00749 | 0,00015 |
| 960 na 544 | 0,04400 | 0,00013 | 0,04412 | 0,01354 | 0,00015 |
| 1280 na 720 | 0,08022 | 0,00014 | 0,07502 | 0,01446 | 0,00016 |

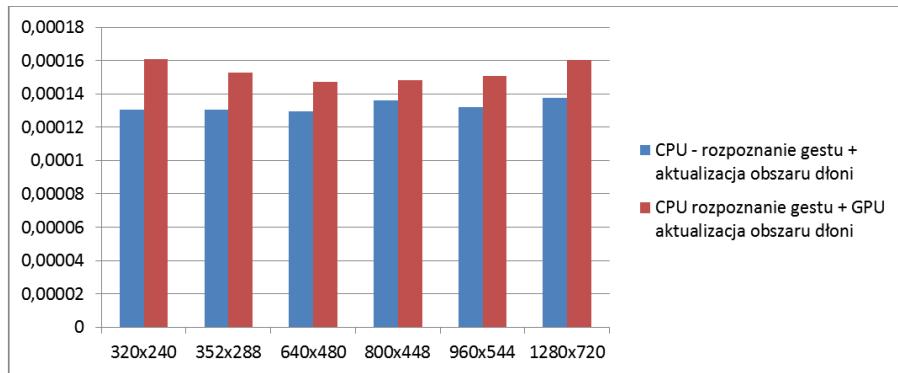
Tabela 4.11. Uśrednione czasy wykonywania całego programu.



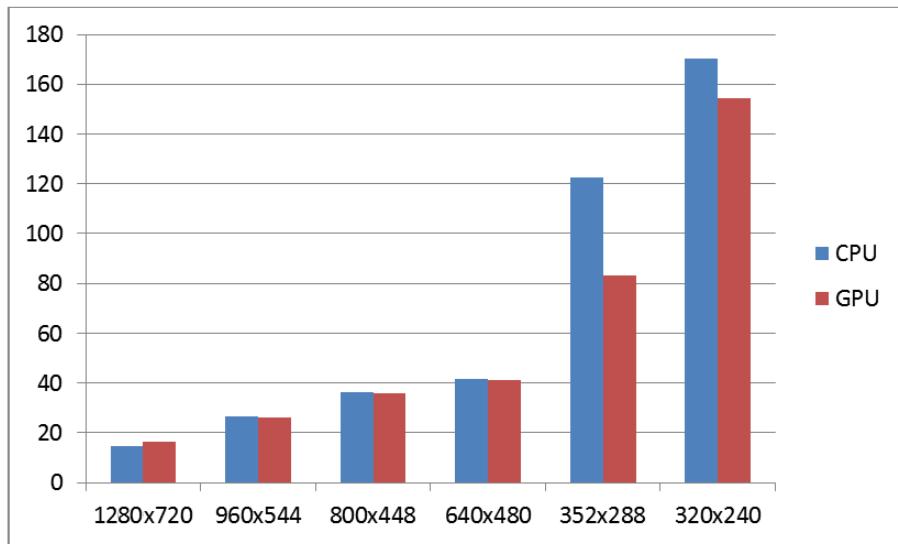
Wykres 4.9. Uśrednione czasy wykonania części programu przed rozpoznaniem gestu.

| FPS | 1920 na 720 | 960 na 544 | 800 na 448 | 640 na 480 | 352 na 288 | 320 na 240 |
|-----|-------------|------------|------------|------------|------------|------------|
| CPU | 14,6 | 26,7 | 36,5 | 41,5 | 123 | 170 |
| GPU | 16,3 | 26,3 | 35,8 | 41,1 | 83,2 | 154 |

Tabela 4.12. Uśredniona liczba klatek na sekundę uzyskiwanych przez program w różnych rozdzielczościach.



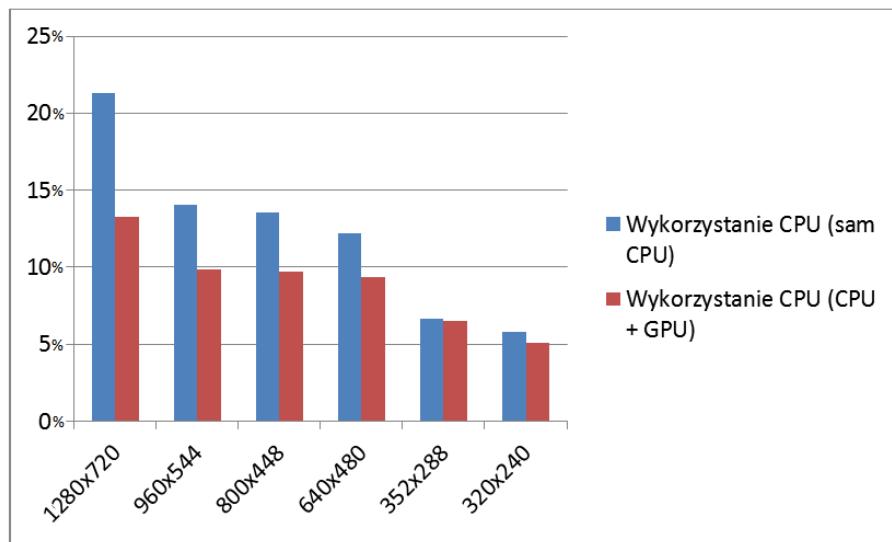
Wykres 4.10. Uśrednione czasy wykonania programu dla części rozpoznania gestu i aktualizacji obszaru dłoni.



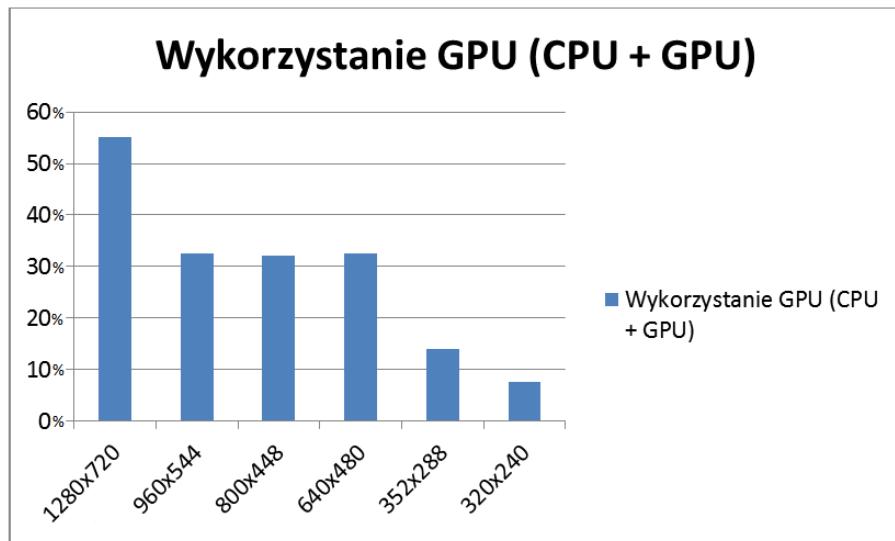
Wykres 4.11. Uśredniona liczba klatek na sekundę uzyskiwane przez program w różnych rozdzielczościach.

| Wykorzystanie | 1920 na 720 | 960 na 544 | 800 na 448 | 640 na 480 | 352 na 288 | 320 na 240 |
|--------------------|-------------|------------|------------|------------|------------|------------|
| CPU (sam CPU) | 21,3 | 14 | 13,6 | 12,2 | 6,64 | 5,82 |
| CPU (CPU + GPU) | 13,3 | 9,86 | 9,73 | 9,38 | 6,5 | 5,08 |
| GPU (CPU + GPU) | 55,2 | 32,6 | 32,1 | 32,6 | 13,9 | 7,56 |

Tabela 4.13. Uśrednione obciążenie zasobów systemowych przy pracy programu w procentach.



Wykres 4.12. Uśrednione obciążenie zasobów systemowych przy pracy programu w procentach.



Wykres 4.13. Uśrednione wykorzystanie procesora graficznego podczas pracy programu w procentach.

tylko na procesorze głównym.

W 57 klatce działania aplikacji nastąpiło pierwsze rozpoznanie dloni na ekranie. Zainicjalizowane zostały punkty na obrazie w miejscu gdzie znajduje się dłoń, dla których następnie obliczany jest przepływ optyczny, który służy do obliczania ruchu dloni. Powoduje to, że w kolejnych ramkach wykonanie programu jest dłuższe o obliczenie przepływu dla wyznaczonych punktów, co jest widoczne w wynikach.

W tabeli 4.11 przedstawiono uśrednione wyniki dla różnych rozdzielczości. Kolumna zatytułowana „GPU - rozpoznanie gestu + aktualizacja obszaru dloni” odnosi się do czasu wykonania algorytmu dopasowania gestu wykonywanego na procesorze graficznym. Do obliczeń został użyty algorytm siłowy (z angielskiego Brute-force), który bada wszystkie elementy ze zbioru treningowego i dopasowuje ten najlepszy zgodnie z pewnym kryterium (zostały analizowane kryterium kraty Manhattan oraz odległość Euklidesowa). Wyniki pokazują, że algorytm ten osiąga wynik dużo gorszy, jeżeli chodzi o szybkość wykonania od implementacji klasyfikatora Bayesa i klasyfikatora k-najbliższych sąsiadów wykonywanych na procesorze głównym. Jako, że wyniki klasyfikacji osiągane przez te drugie są zupełnie zadowalające nie ma potrzeby stosowania dużo wolniejszego algorytmu brutalnej siły. Wyniki zostały zamieszczone w celach poglądowych.

Zgodnie z tabelą 4.11 i wynikami przedstawionymi na wykresie 4.9 widać, że szybkość wykonywania części programu, która w głównej mierze jest wykonywana na procesorze graficznym dla implementacji na GPU nie różni się znacząco od implementacji tej samej części dla CPU. Jest to część programu, która w większej mierze jest wykonywana w asynchronicznym strumieniu CUDATM. Dopiero pod koniec wykonywania następuje pobranie wyników z procesora graficznego i synchronizacja danych. Godna uwagi jest nieznaczna, aczkolwiek widoczna tendencja zmiany czasów wykonywania aplikacji wraz ze wzrostem rozdzielczości. Różnica w czasie wykonywania, z początku faworyzująca rozwiązanie na procesorze CPU, jednak dla rozdzielczości 1280 na 720 pikseli jest ona już korzystna dla rozwiązania zrównoleglonego. Średnio dla tej rozdzielczości różnica ta wynosi około 5 milisekund, co nie jest wartością bardzo znaczącą, lecz z czysto akademickiego podejścia udowadnia postawioną wcześniej tezę, mianowicie, że rozwiązanie na karcie graficznej jest bardziej skalowalne od rozwiązania na procesorze głównym.

Wykres 4.10 ukazuje różnicę pomiędzy wykonywaniem rozpoznania gestu w trakcie przetwarzania aplikacji w trybie z użyciem karty graficznej i bez. Różnica wynika nie ze względu na implementację, gdyż jak zaznaczono wcześniej użyty jest dokładnie ten sam kod do rozpoznania gestu, lecz z czasu wymaganego na wgranie do pamięci procesora graficznego aktualizacji dotyczącej pozycji rozpoznanej dloni potrzebnej do wykrycia ruchu przy analizie kolejnej klatki sekwencji video. Dodatkowo, można zaobserwować, że czas na rozpoznanie gestu nie ulega zmianie przy wzroście rozdzielczości. Spowodowane

jest to faktem, że rozpoznanie gestu jest wykonane na deskryptorach dotyczących analizowanego kształtu, a nie na samych pikselach przedstawiających dłoń. Deskryptory te zostają obliczone we wcześniejszej fazie wykonywania programu.

W kolejnej tabeli 4.12 i wykresie 4.11 przedstawione zostały pomiary ilości klatek analizowanych w przeciągu jednej sekundy w różnych rozdzielczościach. Czas obliczenia ilości klatek nie uwzględnia pobrania obrazu z kamery i ewentualnej prezentacji wyników na ekranie (pokazanie obrazu wraz z naniesionymi obszarami dloni, i innymi danymi dotyczącymi rozpoznawania). Wyniki pokazują, że aplikacja wykonuje się szybciej bez użycia procesora graficznego jedynie dla małych rozdzielczości. Wraz ze wzrostem rozdzielczości liczba klatek spada szybciej dla implementacji na procesor główny.

Ważnym czynnikiem działania aplikacji jest nie tylko liczba klatek na sekundę, ponieważ powyżej pewnej wartości różnica nie jest zauważalna, lecz również zajętość zasobów systemu. W przypadku gdy mamy do czynienia z aplikacją interfejsu użytkownika liczba ta musi być jak najmniejsza. W systemie nie zastosowano żadnego ograniczenia na maksymalną zajętość czasu procesora (ani głównego, ani graficznego), jednak widać wyraźną różnicę wymaganego czasu procesora wraz ze zmianą rozdzielczości.

W tabeli 4.13 i na wykresach 4.12 i 4.13 widać różnicę pomiędzy wymaganymi zasobami dla różnych rozdzielczości. Niewątpliwie widać, szczególnie przy wysokich rozdzielczościach, że implementacja zrównoleglona jest wydajniejsza jeśli chodzi o zużycie zasobów. Dodatkowo należy przypomnieć, że program został testowany na maszynie z procesorem podstawowym posiadającym osiem logicznych wątków przetwarzania (cztery rdzenie po dwa potoki przetwarzania), tak więc mimo, że różnice procentowe nie są znaczne miały by dużo większe znaczenie w przypadku maszyn jedno, bądź dwurdzeniowych.

Aplikacja była testowana na mobilnej karcie graficznej NVIDIA® QUADRO 2000M. Jest to karta dedykowana dla laptopów dlatego jej wydajność jest zdecydowanie mniejsza aniżeli karty zainstalowane w komputerach stacjonarnych. Dlatego też, wyniki dla procesora graficznego osiągane przez aplikację powinny być znaczco lepsze na komputerach stacjonarnych z lepszymi kartami graficznymi.

4.3 Algorytm wydzielania statycznego tła

Potrzeba stworzenia algorytmu wydzielania statycznego tła oraz podstawy działania została przedstawiona w rozdziale 3.2.2. W tym miejscu zostało przedstawione rozwiązanie problemu zaproponowane przez autora. Implementacja jest stworzona zarówno dla przetwarzania równoległego jak i dla sekwencyjnego na procesorze głównym. W kolejnych podrozdziałach zostały przedstawione wymagania i cel, rozwiązanie oraz testy i wyniki badania stworzonego algorytmu.

4.3.1 Wymagania i cel

Głównym przesłaniem stworzenia algorytmu służącego do segmentacji tła była poprawa jakości rozpoznawania skóry na tle, które było podobne do koloru skóry ludzkiej. Mogą to być zarówno drewniane meble jak i ściana pomalowana na żółto, brązowo bądź w kolorze podobnym do skóry.

Głównym wymaganiem jest wysoka wydajność algorytmu, jak również dobre wyniki dla mało dynamicznie zmieniającego się tła. Algorytm powinien dawać poprawne i powtarzalne wyniki niezależnie od obiektów znajdujących się w tle.

Dodatkowo procedura powinna uwzględniać zmianę luminescencji. Jak również całkowitą zmianę tła (na przykład przesunięcie kamery). Algorytm powinien również pozwalać na wydajną implementację na procesorze graficznym.

4.3.2 Zastosowane rozwiązanie

Autorowi udało się osiągnąć wszystkie z postawionych wymagań. Trzeba jednak nadmienić, że nie wszystkie funkcje operujące na obrazie dało się w sposób wydajny zaimplementować na procesorze wielordzeniowym, choć wszystkie z funkcji składowych zostały zaimplementowane w wersji sekwencyjnej jak również równoleglej.

Sam algorytm został podzielony na dwie części. Pierwsza z nich to usuwanie tła z obrazu pobranego z kamery na podstawie wcześniej zgromadzonych informacji dotyczących statycznego tła. Druga część algorytmu to zbieranie i aktualizacja danych dotyczących tła celem późniejszego jego wydzielenia.

Pierwsza część algorytmu została zaimplementowana jako obliczanie różnicy pomiędzy badaną klatką animacji, a zapamiętanymi wcześniej obrazami zawierającymi tło. Operacja usunięcia tła następuje na obrazach w skali szarości. Obliczana jest suma różnic pomiędzy aktualnym obrazem, a trzydziestoma sześcioma obrazami tła. Liczba ta została dobrana empirycznie podczas testów. W zależności od ilości klatek aktualizacja tła następowała w różnym czasie. Im więcej klatek jest używanych do obliczania różnicy tym dłużej następuje ich aktualizacja gdy tło ulegnie zmianie. Z drugiej strony, wraz ze zwiększeniem ilości klatek tła, algorytm jest bardziej odporny na zakłócenia. W trakcie normalnej obsługi programu tło jest relatywnie nie zmienne. Do ustalenia liczby trzydziesięciu sześciu przyczyniły się testy polegające na sprawdzaniu jakości rozpoznawania gestów w przypadku gdy tło ulega nieznacznym zmianom mogącym występować podczas normalnej pracy aplikacji. Testy miały na celu znalezienia ilości klatek przy których program najszybciej reagował na zmiany w otoczeniu obiektu, zachowując tym samym wysoką odporność na zakłócenia.

Następnie obliczona różnica pomiędzy zapamiętanymi obrazami a obecnym tłem jest

progowana, w celu uzyskania obrazu binarnego. Wartość progowa została ustalona na czterdzięcią czyli około piętnasto-procentową różnicą w skali szarości. (Uwaga, zamiast obrazu w skali szarości można było użyć innej przestrzeni barw, jednak chciano uniknąć dodatkowego przekształcenia kolorów, a obraz w skali szarości i tak jest już dostępnny, ponieważ jest wymagany dla algorytmu rozpoznawania twarzy).

Uzyskany w ten sposób obraz binarny jest maską, zawierającą wartość 1 tam gdzie obraz jest tłem, oraz 0 w miejscu gdzie obraz różni się znacząco od tła. Dla usunięcia zakłóceń wykonuje się następnie operacje morfologii (pojedynczą operację otwarcia oraz piętnastokrotną operację zamknięcia - dla rozdzielczości 320 na 240 pikseli, w przypadku większych rozdzielczości wartości te należy przeskalać). Uzyskany w ten sposób obraz jest negowany, a następnie używa się operatora mnożenia logicznego z obrazem wejściowym w celu uzyskania obrazu końcowego zawierającego piksele z wartością 0 zamiast tła.

Drugą częścią algorytmu jest operacja aktualizacji oraz zapisania wartości tła. Część podstawowa, czyli zapisanie wartości aktualnie pobranego obrazu z kamery jako tło jest wykonywana zawsze na początku działania aplikacji oraz na każde życzenie użytkownika (po naciśnięciu przycisku ‘c’). Jest to zwykłe zapisanie całego obrazu w skali szarości do pamięci. Dodatkowo pełna aktualizacja ma miejsce w przypadku, gdy liczba pikseli tła spadnie poniżej 20% całości obrazu (W trakcie normalnie działającego programu użytkownik nie wypełnia swoją dłonią całego kadru kamery). Ma to na celu wymuszenie aktualizacji w przypadku nagłej zmiany oświetlenia lub poruszeniu kamery.

Bardziej zaawansowana jest część aktualizacji w trakcie normalnego działania programu. Algorytm ten jest następujący: aktualizacji podlegają jedynie te piksele, które zostały zaklasyfikowane w danym momencie do tła. Podejście takie pozwala znakomicie wyeliminować liniową zmianę luminescencji, oraz niewielkie zmiany w tle (dzięki operacji zamknięcia w przytoczonym powyżej algorytmie klasyfikacji czy obszar należy do tła czy nie).

Aktualizacji podlega zawsze tylko jedna, kolejna ramka (z trzydziestu sześciu). Podczas kolejnej aktualizacji następuje cykliczne przesunięcie aktualizowanej ramki. Powoduje to zmniejszenie wkradania się błędów do zapamiętywanych obrazów tła. Jedynie w przypadku, gdy zmiana jest stała, zostaje ona rozpropagowana do wszystkich ramek z tłem.

4.3.3 Zrównoleglanie wydzielania tła

Wszystkie wyżej wymienione operacje składowe dla algorytmu wydzielania tła można było zrównoleglić. Jedynie zrównoleglanie operacji obliczenia procentowej zawartości tła

| | CPU | GPU |
|-------------|--------|--------|
| 320 na 240 | 0,0025 | 0,0038 |
| 352 na 288 | 0,0033 | 0,0043 |
| 800 na 448 | 0,0115 | 0,0127 |
| 960 na 544 | 0,0170 | 0,0178 |
| 1280 na 720 | 0,0297 | 0,0293 |

Tabela 4.14. Uśrednione czasy wykonania algorytmu segmentacji tła.

| | CPU | GPU |
|-------------|--------|--------|
| 320 na 240 | 0,0002 | 0,0012 |
| 352 na 288 | 0,0003 | 0,0014 |
| 800 na 448 | 0,0010 | 0,0017 |
| 960 na 544 | 0,0012 | 0,0020 |
| 1280 na 720 | 0,0022 | 0,0021 |

Tabela 4.15. Uśrednione czasy wykonania aktualizacji zapamiętanego tła dla algorytmu segmentacji tła.

w obrazie nie niosła za sobą jawnych korzyści, ponieważ w celu obliczenia tej wartości niezbędne jest zsumowanie wszystkich pikseli znajdujących się na obrazie. Skutkuje to tym, że niemożliwa jest dekompozycja problemu na pojedyncze piksele.

Do zrównoleglenia użyto implementacji funkcji na procesorze graficznym znajdującej się w bibliotece OpenCV. Dodatek B zawiera implementację ważniejszych funkcji w algorytmie.

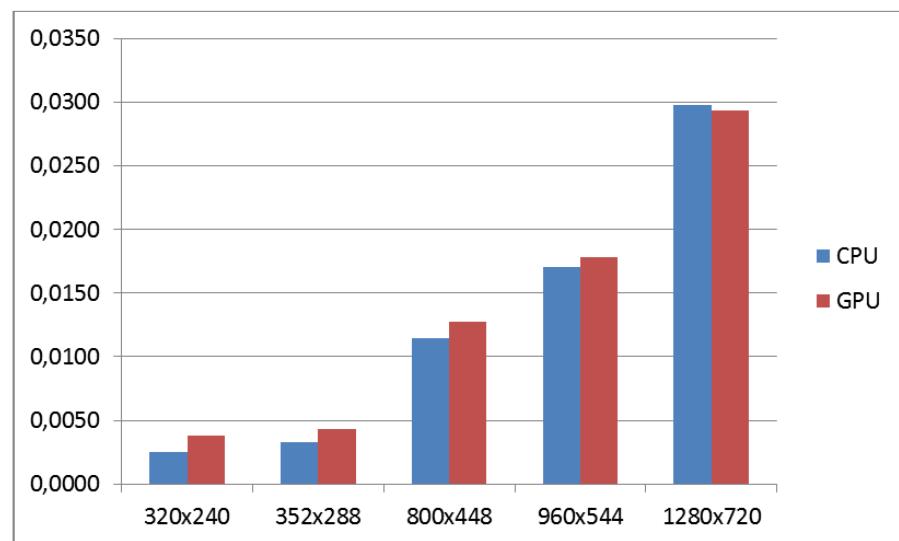
4.3.4 Badania rozwiązań i wyniki

W celu zbadania algorytmu zostały przeprowadzone testy wydajnościowe dla różnych rozdzielczości. Przedstawione w pracy wyniki są danymi średnimi. Algorytm składa się z dwóch części. W tabeli 4.14 i na wykresie 4.14 zostały przedstawione wyniki z części aktualizującej dane na temat tła (część ta jest dużo szybsza).

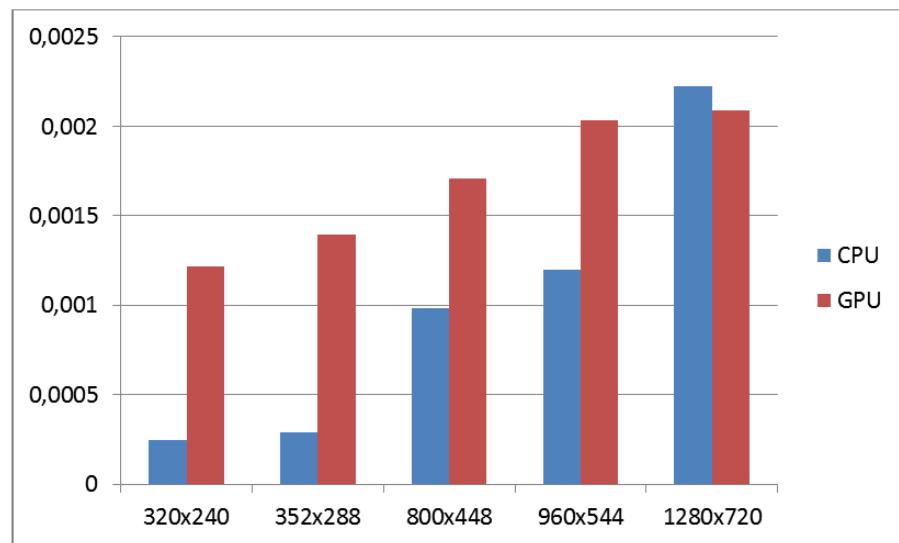
Tabela 4.15 i 4.15 przedstawiają czasy zapamiętania statycznych pikseli dla algorytmu segmentacji tła.

4.3.5 Wnioski

Podobnie jak w przypadku innych testowanych algorytmów, algorytm segmentacji tła daje bardzo przybliżone czasy wykonania dla implementacji dla procesora centralnego



Wykres 4.14. Uśrednione czasy wykonania algorytmu segmentacji tła.



Wykres 4.15. Uśrednione czasy wykonania aktualizacji zapamiętanego tła dla algorytmu segmentacji tła.

jak również dla zrównoległej wersji na procesor graficzny. Również i w tym przypadku widać, że dla wyższych rozdzielczości wydajniejszą okazuje się implementacja dla procesora graficznego. Warto jednak zaznaczyć, że różnice te nie są znaczące.

4.4 Algorytm wydzielania dloni z całej ręki

Jednym z celów pracy jest ulepszenie funkcjonowania aplikacji stworzonej w trakcie pracy inżynierskiej przez autora. Mimo, że skupiono się przede wszystkim na zrównolegleniu aplikacji, dodano również nowe funkcjonalności. Jedną z takich funkcjonalności jest algorytm wydzielania obszaru dloni z całej ręki.

Algorytm ten pozwala na pozbycie się bardzo uciążliwego ograniczenia, że użytkownik musi nosić ubranie z długim rękawem w celu poprawnego rozpoznawania dloni. W pracy został zaimplementowany algorytm pozwalający na usunięcie tej niedogodności.

4.4.1 Wymagania i cel

Wymaganiem jest stworzenie algorytmu potrafiącego z dowolnego obszaru wydzielić obszar dloni. Algorytm zakłada, że dłoń będzie pokazywana palcami do góry oraz wewnętrzna stroną w kierunku kamery.

Celem stworzenia tego algorytmu jest, jak już wspomniano wcześniej, pozbycie się ograniczenia wymuszającego na użytkowniku systemu noszenia ubrania o długich rękawach w kolorze odróżniającym się od skóry ludzkiej.

4.4.2 Zastosowane rozwiązanie

Algorytm w systemie jest wykonywany po rozpoznaniu obszarów będących skórą ludzką. Badane są wszystkie obszary mogące potencjalnie być dlonią. Biorąc pod uwagę niedoskonałości algorytmów segmentacji skóry mogą pojawić się obszary przedstawiające dłoń, lecz przeważnie znaczaco różnią się one od kształtu dloni i mogą zostać wyeliminowane.

Dla każdego z badanych obszarów wykonywane są kolejno następujące kroki:

1. Brany do przeanalizowania jest kolejny obszar z listy rozpoznanych obszarów (w przypadku dojścia do końca listy, algorytm jestkończony).
2. Do badanego obszaru dopasowywana jest elipsa za pomocą funkcji z biblioteki OpenCV.
3. Jeżeli obszar elipsy jest zbyt mały algorytm jest kontynuowany dla kolejnego obszaru (powrót do punktu 1).

4. Obszar jest następnie obracany aby dłuższa przekątna elipsy była pionowo względem układu osi współrzędnych.
5. Obliczany jest prostokąt opisany na obszarze (warto zaznaczyć, że wcześniej dopasowywano elipę, jednak było to najlepsze dopasowanie średnie, a nie elipsa opisana na obiekcie).
6. Ustalona zostaje maksymalna wysokość wyjściowa obszaru jako 120% szerokości opisanego prostokąta obliczonego w punkcie 5. Jest to innymi słowy, 1,2 razy większa odległość niż odległość pomiędzy najbardziej wysuniętymi na lewo i prawo punktami obszaru czyli szerokością dloni.
7. Obszar zostaje przycięty od dołu do maksymalnej wysokości obliczonej w punkcie 6.
8. Powrót do punktu 1.

Algorytm został zaimplementowany jedynie w wersji sekwencyjnej dla procesora głównego. Spowodowane jest to faktem, że ilość danych na tym etapie przetwarzania jest już znacznie zredukowana i zrównoleglenie nie przyniosłoby żadnych korzyści. Narzut na komunikację pomiędzy procesorem głównym, a procesorem graficznym byłby większy niż zysk otrzymany ze zrównoleglenia.

Procedura została przetestowana dla rzeczywistych danych i osiąga bardzo dobre wyniki dla większości przypadków. Algorytm jednak nie jest w stanie poprawnie wydzielić obszaru dloni, gdy rozpoznany obszar skóry uwzględnia większą część sylwetki ludzkiej obejmującą tors i głowę (jako jeden obszar), a nie samą rękę. Przy normalnym użytkowaniu aplikacji, problem ten nie występuje i został pominięty, jako mało znaczący.

Algorytm wydzielania dloni został z sukcesem włączony do implementacji programu poprawiając działanie końcowe aplikacji.

4.5 Sieciowy interfejs programu

Tworzona aplikacja służy jako interfejs użytkownika. Pozwala ona na sterowanie myszą komputerową za pomocą dloni. Jest to jedna z wielu możliwości wykorzystania potencjału znajdującego się w aplikacji.

Dla stworzenia jak najbardziej uniwersalnej komunikacji aplikacji z zewnętrznymi systemami stworzono interfejs sieciowy pozwalający na komunikację z dowolnym urządzeniem posiadającym obsługę sieci i implementującym zaproponowany model komunikacji. W tym celu stworzono interfejs oparty na powszechnie znanych gniazdach sieciowych oraz

stworzono przykładową aplikację kliencką odbierającą informacje o rozpoznanych przez aplikację serwera gestach i ruchu dloni.

4.5.1 Opis zastosowanego rozwiązania

Komunikacja pomiędzy aplikacją serwera wysyłającą informacje o rozpoznanych gestach i ruchu dloni (dalej zwaną serwerem), a aplikacją kliencką odbierającą te informacje (dalej zwaną klientem) odbywa się poprzez wysyłanie paczek danych o ścisłe ustalonym formacie (opisanych dalej). Aby wysyłanie to było możliwe, musi zostać nawiązane połączenie pomiędzy serwerem, a klientem (bądź wieloma klientami).

Algorytm połączenia wygląda następująco:

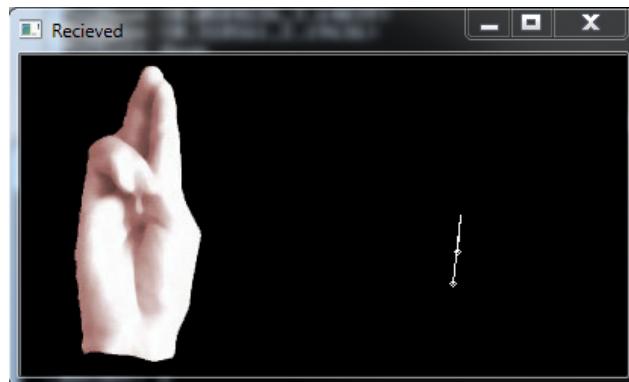
- Tworzony jest serwer przez aplikację analizującą obrazy z kamery na porcie nr 8879 oraz rozpoczęte jest nasłuchiwanie czy jakiś klient nie próbuje ustanowić połączenia. (Uwaga. Port 8879 jest umowny i został przyjęty na potrzeby aplikacji).
- Klient wysyła, żądanie na port nr 8879 celem ustanowienia połączenia z serwerem. Jeżeli serwer istnieje połączenie zostaje nawiązane, jeżeli nie to próba połączenia nastąpi za kolejne 5 sekund i tak aż do skutku.
- Po ustanowieniu połączenia wszystkie wyniki rozpoznawania zostają przesyłane do każdego połączonego klienta.

Poniżej opisano wszystkie możliwe pakiety danych wysyłanych przez serwer:

- Pakiet zawierający jeden znak bajtowy typu char o kodzie 1 oznacza gest R.
- Pakiet zawierający jeden znak bajtowy typu char o kodzie 2 oznacza gest A.
- Pakiet zawierający jeden znak bajtowy typu char o kodzie 3 oznacza gest T.
- Pakiet zawierający jeden znak bajtowy typu char o kodzie 4 oznacza gest Otwarta dłoń.
- Pakiet zawierający jeden znak bajtowy typu char i kodzie 5 oznacza zwiększenie czułości ruchu (użytkownik nacisnął klawisz + będąc w aplikacji serwera).
- Pakiet zawierający jeden znak bajtowy typu char i kodzie 6 oznacza zmniejszenie czułości ruchu (użytkownik nacisnął klawisz – będąc w aplikacji serwera).
- Pakiet zawierający jeden znak bajtowy typu char o kodzie 0, oraz dwa razy wartość typu double oznaczającą ruchu ręki o interwał w poziomie i pionie przekazanych odpowiednio w pierwszej liczbie double po bajcie sterującym typu char oraz drugiej liczbie double. Pakiet ten został przedstawiony w tabeli 4.16.

| Bit początkowy | Typ danych | Informacja |
|----------------|-------------------|-----------------------|
| 0 | char (8 bitów) | kod sterujący równy 0 |
| 8 | double (64 bitów) | interwał w poziomie |
| 72 | double (64 bitów) | interwał w pionie |

Tabela 4.16. Pakiet zawierający informację o ruchu.



Zrzut z ekranu 4.16. Przykładowy program kliencki.

4.5.2 Przykładowy klient

Do celów prezentacyjnych zaimplementowano przykładową aplikację kliencką. Wyświetla ona na ekranie gest odpowiadający odebranemu przez sieć pakietowi oraz pokazuje wektor ruchu dłoni. Dodatkowo ruch jest przekładany na ruch kurSORA na ekranie. Przykładowa aplikacja kliencka jest widoczna na zrzucie ekranu 4.16.

Warto zaznaczyć, że aplikacją kliencką może być każda inna aplikacja na przykład do sterowania trójwymiarowym programem graficznym. Integracja ta jest bardzo efektywna, a przy tym możliwe najprostsza w implementacji.

Rozdział 5

Podsumowanie

Reasumując, udało się osiągnąć wszystkie postawione w pracy cele. Program działa wydajniej niż jego poprzednia wersja napisana w trakcie pracy inżynierskiej. Jednocześnie uległa poprawie poprawność rozpoznawania gestów i ruchu dloni. Dzięki przeprowadzonym eksperymentom można jasno stwierdzić, że implementacja systemu uległa poprawie. Zostało to osiągnięte dzięki zmianie wielu poszczególnych algorytmów (w tym algorytm wydzielania tła, algorytm wydzielania dloni), jak również zmianie sposobu obliczeń poprzez użycia procesora graficznego do przetwarzania danych. Poniżej przedstawiono wyniki osiągnięte w trakcie tworzenia pracy.

Z sukcesem osiągnięto lepszy poziom dokładności w trafnym rozpoznaniu gestu dloni oraz lepsze i płynniejsze przełożenie ruchu dloni na dane w programie. Na poprawność rozpoznania złożyło się wiele czynników poprawiających jakość działania algorytmu dopasowania kształtu do wzorca poprzez dokładniejsze wydzielenie obszaru dloni z analizowanego obrazu. Przykładowo w pierwszej wersji systemu niemożliwe było działanie aplikacji gdy na tle znajdowały się elementy drewniane (kolorem przypominające skórę). W obecnej wersji systemu działanie programu nie zostaje zakłócone i rozpoznawanie odbywa się bez przeskódek.

Na poprawienie wydzielania obszaru dloni z obrazu duży wpływ ma nowy, autorski algorytm wydzielania statycznego tła w obrazie. Pozwala on na bardziej precyzyjne wydzielenie tła, aniżeli miało miejsce to w przypadku algorytmu zaimplementowanego w poprzedniej wersji systemu. Algorytm nie wnosi również żadnych zniekształceń w obszarze dloni. Zastosowanie algorytmu pozwoliło na zwiększenie zakresu pracy programu. Zrównoleglenie algorytmu segmentacji tła nie przyniosło wymiernych korzyści w czasie wykonywania obliczeń.

Dodatkowo zaimplementowany został algorytm pozwalający wyeliminować wymaganie posiadania przez użytkownika ubrania z długim rękawem (w celu jasnego oddzielenia

dłoni od przedramienia). Dla osiągnięcia powyższego stworzono procedurę wydzielającą obszar dłoni z rozpoznanego kształtu. Implementacja algorytmu pozwoliła na rozszerzenie funkcjonalności programu oraz spowodowała, że system stał się bardziej przyjazny użytkownikowi.

Dzięki zastosowaniu technologii NVIDIA® CUDA™ udało się ograniczyć zużycie zasobów systemowych przez program utrzymując jednocześnie szybkość działania aplikacji. Wbrew oczekiwaniom program nie działa szybciej dla rozdzielczości przetwarzania 320 na 240 pikseli. Przy takiej rozdzielczości system przetwarza około sto pięćdziesiąt cztery klatek na sekundę działając z pomocą procesora graficznego i około sto siedemdziesiąt klatek na sekundę działając jedynie na procesorze głównym. Ogólny wzrost szybkości działania zauważalny jest dopiero przy rozdzielczości 1280 na 720. Przy takiej rozdzielczości system przetwarza około czternastu klatek na sekundę bez użycia procesora graficznego i szesnastu klatek na sekundę w przypadku użycia obu procesorów. Niemniej jednak, użycie procesora graficznego w przetwarzaniu przyniosło wymierne korzyści, co potwierdziło słuszność tezy postawionej na początku pracy. Główną z korzyści jest oddalenie systemu operacyjnego dzięki mniejszej zajętości czasu głównego procesora przez aplikację. Przykładowo dla rozdzielczości 1280 na 720 jest to redukcja z 22% do 13% wykorzystania mocy obliczeniowej czterordzeniowego procesora.

Mimo ogólnego braku przyspieszenia wykonywania całości programu należy zaznaczyć, że niektóre jego części uległy znacznemu przyspieszeniu (nawet kilkunastokrotnemu). Taką częścią programu jest część rozpoznająca kolor skóry i wydzielająca obszary w obrazie, na których potencjalnie może znajdować się skóra ludzka. Przy rozdzielczości 320 na 240 pikseli przyspieszenie jest pięciokrotne, ale dla rozdzielczości 1280 na 720 pikseli przyspieszenie jest już szesnastokrotne. Wszystkie dziesięć algorytmów z pracy inżynierskiej zostało zrównoleglonych i przepisanych w celu umożliwienia ich wykonania na procesorze graficznym. Algorytm ten jest jednak jedynie częścią całości przetwarzania obrazów, dlatego w ogólnym wyniku szybkości działania systemu nie jest zauważalna.

Pośredni wniosek płynący z pracy dotyczy przetwarzaniu równoległemu jako ogólnej metodzie optymalizacji czasu wykonywania algorytmów. Wniosek, zgodny z prawem Amdahla, jest następujący: zrównoleglenie metody ma sens tylko wtedy, jeżeli część kodu mogącego zostać wykonywana w wielu potokach przetwarzania jest duża w stosunku do całości obliczeń. Pośrednio ma to również wpływ na ilość danych wejściowych algorytmu. Jeżeli ta ilość jest spora, a łatwo można dokonać ich dekompozycji, wtedy korzyść płynąca z przetwarzania równoległego jest znaczna.

W pracy zaimplementowano interfejs sieciowy pozwalający na odbiór przeanalizowanych danych przez inne programy. Pozwala to na stworzenie zewnętrznych systemów opartych na programie służącym do rozpoznawania dloni. Dzięki zastosowaniu takiego

rozwiązań możliwa jest praktycznie nieograniczona rozbudowa systemu i dopasowanie go do indywidualnych potrzeb użytkowników końcowych. Dodatkową zaletą jest możliwość używania systemu zdalnie przy użyciu sieci komputerowej.

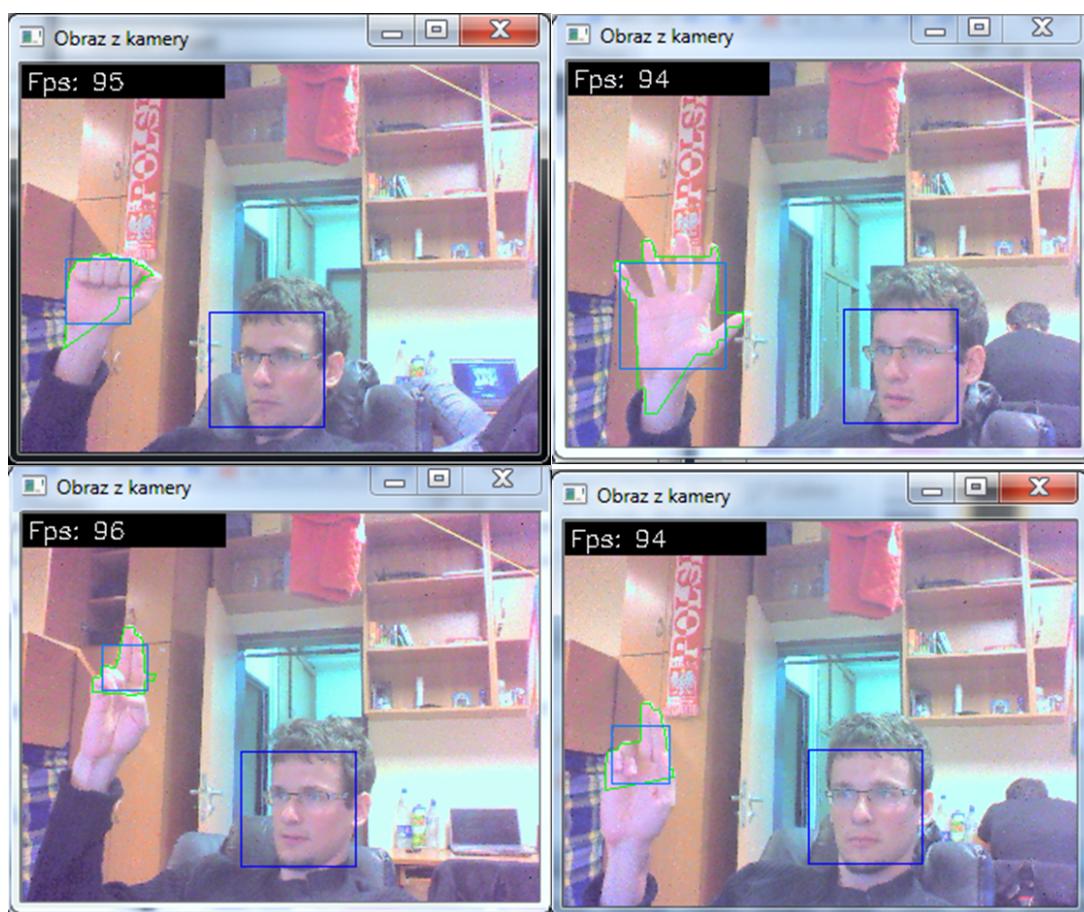
Bibliografia

- [1] C. Urdiales i F. Sandoval A. Domínguez-Caneda. Dynamic background subtraction for object extraction using virtual reality based prediction. *Universidad de Málaga, Spain*, 2006.
- [2] Alfredo Petrosino Alain Merigot. Parallel processing for image and video processing: Issues and challenges. *Université Paris Sud, Orsay, France oraz University of Naples "Parthenope", Naples, Italy*, 2008.
- [3] Joe Armstrong. Making reliable distributed systems in the presence of software errors. *The Royal Institute of Technology Stockholm, Sweden*, 2003.
- [4] Jean-Yves Bouguet. Pyramidal implementation of the lucas kanade feature tracker description of the algorithm. *Intel Corporation Microprocessor Research Labs*.
- [5] A. Valli C. Colombo, A. Del Bimbo. A real-time full body tracking and humanoid animation system. *Università di Firenze, Italy*, 2008.
- [6] NVIDIA CUDATM. Nvidia cuda c programming guide - version 4.2. WWW.
- [7] Frederica Darema. *The SPMD Model : Past, Present and Future*. Lecture Notes in Computer Science, 2001.
- [8] B Michael Bove Jr. Darren Butlelel, Sridha Sridharun. Real-time adaptive background segmentation. *Queensland University of Technology, Cambridge MA 02139*, 2003.
- [9] Domenico Tegolo Francesco Isgro. A distributed genetic algorithm for restoration of vertical line scratches. *Università di Napoli Federico II, Universita di Palermo*, 2008.
- [10] Brian "Beej" Hall. *Beej's Guide to Network Programming*. beej@piratehaven.org, 1995-2001.
- [11] <http://openmp.org>. Openmp - oficjalna strona projektu. WWW.

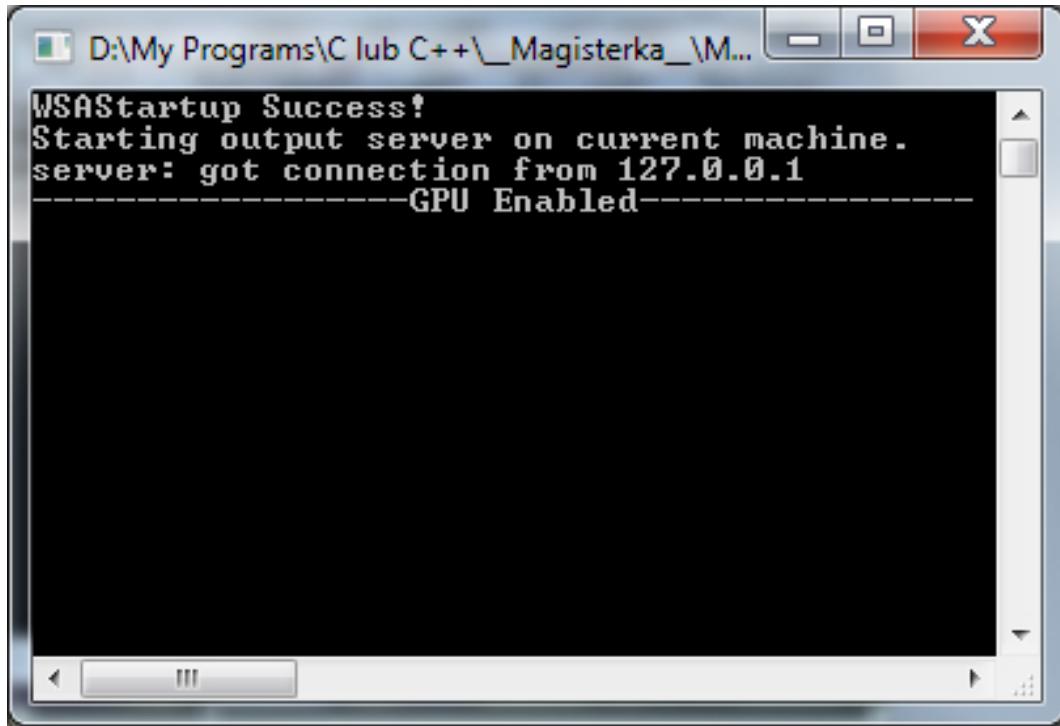
- [12] <http://setiathome.berkeley.edu/>. Oficjalna strona projektu seti @ home. WWW.
- [13] http://software.intel.com/en-us/intel_mkl. Intel mkl - oficjalna strona projektu. WWW.
- [14] <http://www.khronos.org/opencl/>. Opencl - oficjalna strona projektu. WWW.
- [15] Joe Mambretti. The grid and grid network services.
- [16] Daniel Minoli. *A network approach to grid computing*. John Wiley and Sons, Inc., 2005.
- [17] Steve Rennich NVIDIA. Cuda c/c++ streams and concurrency. WWW - prezentacja.
- [18] C. Nicolescu P.P. Jonker, J.G.E. Olk. Distributed bucket processing: A paradigm embedded in a framework for the parallel processing of pixel sets. *Delft University of Technology, Netherlands*, 2008.
- [19] Przemysław Korohoda Ryszard Tadeusiewicz. *Komputerowa analiza i przetwarzanie obrazów*. Wydawnictwo Fundacji Postępu Telekomunikacji Kraków, 1977.
- [20] New Scientist. Kinect imaging lets surgeons keep their focus. *19 maj, str. 19*, 2012.
- [21] Filipe Tomaz, Tiago Candeias, and Hamid Shahbazkia. Improved automatic skin detection in color images. *Universidade do Algarve, FCT, Campus de Gambelas - 8000 Faro, Portugal*, 2003.
- [22] Damian Turczyński. Interfejsy użytkownika bazujące na rozpoznawaniu ruchu i gestów dloni (praca dyplomowa inżynierska). *Politechnika Warszawska, wydział Elektroniki i Technik Informacyjnych*, 2010.
- [23] Keita Takahashi i Takeshi Naemura Viet-Quoc Pham. Foreground-background segmentation using iterated distribution matching. *The University of Tokyo*.
- [24] GU Ping-ping YUAN Hong. Application of windows inter-process communication in software system integration. *Nanjing University of Technology*, 2010.

Dodatek A

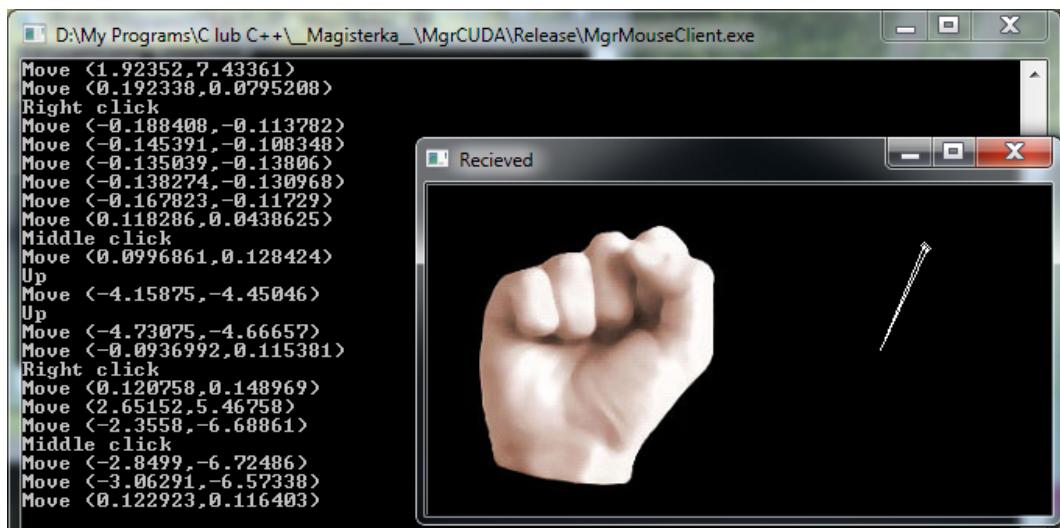
Zrzuty ekranu działającej aplikacji



Zrzut z ekranu A.1. Obraz z kamery z naniesioną pozycją dłoni (zielona obwódka), obszarem dłoni (jasnoniebieski prostokąt) i znalezioną twarzą (ciemnoniebieski prostokąt).



Zrzut z ekranu A.2. Konsola programu serwera.



Zrzut z ekranu A.3. Program kliencki wraz z konsolą i rozpoznanymi gestami oraz ruchem dłoni.

Dodatek B

Kod źródłowy rozwiązań autorskich

B.1 Pierwszy krok zdekomponowanej erozji binarnej na procesorze graficznym.

```
--global__ void gpuFastBinaryErodeStep1(
    const cv::gpu::DevMem2D<unsigned char> src,
    cv::gpu::DevMem2D<unsigned char> dst, const int iterations) {
const int x = blockDim.x * blockIdx.x + threadIdx.x;
const int y = blockDim.y * blockIdx.y + threadIdx.y;

if (x < src.cols && y < src.rows){
    for(int i = -iterations; i <= iterations; ++i){
        if(x + i > 0 && x + i < src.cols){
            if(src.ptr(y)[x+i] != 255){
                dst.ptr(y)[x] = 0;
                return;
            }
        }
    }
    dst.ptr(y)[x] = 255;
}
}
```

B.2 Drugi krok zdekomponowanej erozji binarnej na procesorze graficznym.

```
--global__ void gpuFastBinaryErodeStep2(
    const cv::gpu::DevMem2D<unsigned char> src,
    cv::gpu::DevMem2D<unsigned char> dst, const int iterations) {
const int x = blockDim.x * blockIdx.x + threadIdx.x;
const int y = blockDim.y * blockIdx.y + threadIdx.y;
if (x < src.cols && y < src.rows){
    for(int i = -iterations; i <= iterations; ++i){
        if(y + i > 0 && y + i < src.rows){
            if(src.ptr(y+i)[x] != 255){
                dst.ptr(y)[x] = 0;
                return;
            }
        }
    }
    dst.ptr(y)[x] = 255;
}
}
```

B.3 Filtr wstępny segmentacji koloru skóry.

```
--global__ void gpuInitialFilter(cv::gpu::DevMem2D<Pixel> src) {
const int x = blockDim.x * blockIdx.x + threadIdx.x;
const int y = blockDim.y * blockIdx.y + threadIdx.y;
if (x < src.cols && y < src.rows){
    Pixel px = src.ptr(y)[x];
    float r = px.r;
    float g = px.g;
    float b = px.b;

    float sum = r + g + b;
    if( (b > 160 && r < 180 && g < 180) || //Too much blue
        (g > 160 && r < 180 && b < 180) || //Too much green
        (b < 70 && r < 70 && g < 70) || //Too dark
        (r+g > 400 && b< 170) ||//Too much red and green(yellow like color)
```

```

        (b/(sum) > .4) || //Too much blue in contrast to others
        (g/(sum) > .4) //Too much green in contrast to others
        || (r<240 && g<240 && b<240 && abs(r-g)<10 && abs(g-b)<20
            && abs(r-b) < 30 && r>200)
        || (r<240 && g<240 && b<240 && abs(r-b)<5)
    ){

        Pixel black;
        black.r = 0;
        black.g = 0;
        black.b = 0;
        src.ptr(y)[x] = black;
    }
}

}
}

```

B.4 Jedna z funkcji segmentacji skóry.

```

__global__ void gpuYCbCrFunction(const cv::gpu::DevMem2D<Pixel> src,
                                  cv::gpu::DevMem2D<Pixel> dst) {
    const int x = blockDim.x * blockIdx.x + threadIdx.x;
    const int y = blockDim.y * blockIdx.y + threadIdx.y;
    if (x < src.cols && y < src.rows){
        Pixel px = src.ptr(y)[x];
        float r = px.r;
        float g = px.g;
        float b = px.b;
        float cb = -0.1687f*r - 0.3312f*g + 0.500f*b + 128.0f;
        float cr = 0.500f*r - 0.4183f*g - 0.0816f*b + 128.0f;
        FIT_RANGE(cb, 0.0, 255.0);
        FIT_RANGE(cr, 0.0, 255.0);
        float sum = r+g+b;
        if ( r>95&&g> 40 && b > 20
            && MAX(MAX(r,g),b)-MIN(MIN(r,g),b) > 15
            && abs(r-g) > 15 && r > g && r > b
            && (b/g<1.249)&& b/g > 0.5 &&
            && (sum/(3*r)>0.692)
            && (0.3333-b/sum>0.029)

```

```

    && (g/(3*sum)<0.124)
    || (3*b*r*r)/(sum*sum*sum) >0.110
    && ((r*b + g*g ) / g*b) > 5000
    && sum/(3*r + (r-g)/sum) < 2.7775
    && cb >= range_cb_min
    && cb <= range_cb_max
    && cr >= range_cr_min
    && cr <= range_cr_max
){
    Pixel white; white.r = 255; white.g = 255; white.b = 255;
    dst.ptr(y)[x] = white;
}else{
    Pixel black; black.r = 0; black.g = 0; black.b = 0;
    dst.ptr(y)[x] = black;
}
}
}
}

```

B.5 Algorytm segmentacji statycznego tła z obrazu.

```

void Calibration::clearBackground(
                                const Mat& frame, Mat& afterProcessingImage){

    Mat grayFrame;
    cvtColor(frame, grayFrame, CV_BGR2GRAY);
    if(!calibrationImagesFilled){
        if(tempMat.empty()){
            differResult = Mat(grayFrame.rows, grayFrame.cols, grayFrame.type());
            tempMat = Mat(grayFrame.rows, grayFrame.cols, grayFrame.type());
        }
        calibrationImages[currentCalibrationImageIndex++] = grayFrame.clone();

        if(currentCalibrationImageIndex == numberOfWorkingImages){
            calibrationImagesFilled = true;
            currentCalibrationImageIndex = 0;
        }
    }else{
        for(int i = 0; i < numberOfWorkingImages; ++i){

```

```
absdiff(calibrationImages[i], grayFrame, tempMat);
if(i > 0){
    addWeighted(differResult, 1, tempMat, calibrationImagesDivider,
                0, differResult);
} else{
    addWeighted(differResult, 0, tempMat, calibrationImagesDivider,
                0, differResult);
}
currentCalibrationImageIndex = (currentCalibrationImageIndex + 1) %
                                numberOfCalibrationImages;

threshold(differResult, differResult, thresholdLevel, 255,
          CV_THRESH_BINARY);

morphologyEx(differResult, differResult, MORPH_OPEN, Mat(),
              Point(-1,-1), 1);
morphologyEx(differResult, differResult, MORPH_CLOSE, Mat(),
              Point(-1,-1), 15);
cvtColor(differResult, afterProcessingImage, CV_GRAY2BGR);

//color inversion for background update
bitwise_not(differResult, differResult);

//clear the background
bitwise_and(frame, afterProcessingImage, afterProcessingImage);

//update static backgorund
int nonZeroPixels = countNonZero(differResult);

if(refreshBackground == 0 && nonZeroPixels <
   (differResult.cols*differResult.rows / 5)){
    refreshBackground = numberOfCalibrationImages;
}

if(refreshBackground > 0){
    calibrationImages[currentCalibrationImageIndex] = grayFrame.clone();
```

```

        --refreshBackground;
    }else{
        grayFrame.copyTo(calibrationImages[currentCalibrationImageIndex],
                          differResult);
    }
}
}
}

```

B.6 Algorytm wydzielenia dloni z obszaru.

```

void HandFinder::correctHandArea(vector<vector<Point> >& contours ){
    //find extremums
    for(vector<vector<Point> >::iterator contourIt = contours.begin();
        contourIt!=contours.end(); ++contourIt){
        //fitEllipse
        if(contourIt->size() <= 5){
            continue;
        }
        RotatedRect r = fitEllipse(*contourIt);

        //Get rid of to small objects!
        if(r.size.width * r.size.height < 1000){
            continue;
        }

        Mat m = getRotationMatrix2D(Point(0,0), r.angle - 180, 1);
        Mat minv = getRotationMatrix2D(Point(0,0), -r.angle + 180, 1);
        transform(*contourIt, *contourIt, m);

        Rect rect = boundingRect(*contourIt);
        int maxHeight = (int)(rect.width * 1.2);
        int maxY = rect.tl().y + maxHeight;

        Point previousPt = *(contourIt->rbegin());
        if(maxHeight < rect.height){
            for(vector<Point>::iterator pointIt = contourIt->begin();
                pointIt != contourIt->end(); ){

```

```
if(pointIt->y > maxY){
    if(previousPt.y > maxY){
        pointIt = contourIt->erase(pointIt);
        if(pointIt == contourIt->end()){
            break;
        }
        previousPt = *pointIt;
        continue;
    }else{
        //previous good
        pointIt->x = previousPt.x + (maxY - previousPt.y)*(pointIt->x -
            previousPt.x)/(pointIt->y - previousPt.y);
        pointIt->y = maxY;
    }
}else{
    if(previousPt.y > maxY){
        pointIt->x += (maxY - pointIt->y)*(previousPt.x - pointIt->x) /
            (previousPt.y - pointIt->y );
        pointIt->y = maxY;
    }//else this is good
}
previousPt = *pointIt;
++pointIt;
}
}
transform(*contourIt, *contourIt, minv);
}
}
```

Dodatek C

OpenCV oraz NVIDIA® CUDA™ Przegląd i zastosowania praktyczne

C.1 Wstęp

W pracy zostały omówione kolejno biblioteka programistyczne OpenCV, NVIDIA® CUDA™, połączenie dwóch powyższych, jak również przedstawione zostały inne powiązane rozwiązania.

Drugi rozdział traktujący o bibliotece programistycznej OpenCV został uformowany następująco: w pierwszej części zostały przedstawione główne jej moduły, następnie zostały podane przykłady oraz zastosowania. Dla lepszego zrozumienia rozwiązania został przedstawiony prosty kod źródłowy napisany w języku C++ z użyciem biblioteki OpenCV.

Następnie przedstawione zostało rozwiązanie NVIDIA® CUDA™, jej główne cechy i charakterystyka. Omówione zostały główne interfejsy programistyczne. Na podstawie działania programu autora przedstawione zostały wyniki przyspieszenia działania programu dzięki zastosowaniu NVIDIA® CUDA™. Na podstawie kodu źródłowego podane zostały przykłady działania i zastosowania w praktyce.

W czwartym rozdziale została opisana możliwość połączenia powyższych rozwiązań oraz korzyści z tego płynące. Przedstawiony został również kod źródłowy przykładowego programu używającego powyższych technologii oraz został on omówiony.

Ostatni rozdział jest zarówno podsumowaniem, jak również wspomina o innych rozwiązańach problemu zrównoleglania kodu przy użycia procesora graficznego i głównego procesora CPU.

C.2 OpenCV

C.2.1 Główne cechy

OpenCV jest biblioteką programistyczną powstałą w 2006 roku (wersja 1.0). Prace nad nią zostały rozpoczęte już w roku 1999. Głównymi inicjatorami biblioteki była firma Intel® oraz Willow Garage. Obecna wersja biblioteki to 2.3.1, jest jednak ona wciąż rozwijana i w krótkim okresie czasu powinna powstać wersja 2.4. Na dzień dzisiejszy biblioteka jest przeznaczona dla języków C, C++, Python. Wkrótce ma zostać wypuszczona wersja JAVA. OpenCV jest biblioteką wieloplatformową obsługiwana przez system Linux, Windows i Android. Została wydana na otwartej licencji BSD.

Biblioteka OpenCV jest bardzo znaną biblioteką umożliwiającą przetwarzanie obrazów oraz udostępniającą wiele zaawansowanych algorytmów z zakresu robotyki, uczenia maszynowego jak i innych (w tym matematycznych). Jest ona rozpowszechniona i rozpoznawalna przez programistów na całym świecie (ponad 40 000 aktywnych użytkowników na grupach dyskusyjnych). Implementacja biblioteki zawiera ponad 2500 różnych algorytmów [1].

C.2.2 Moduły

Moduły OpenCV zostały podzielone następująco:

- core – główna funkcjonalność, funkcje podstawowe, główne struktury danych w tym Mat – klasa reprezentująca obraz, jak również wszelkie dane tabelaryczne i wieloprzestrzenne.
- imgproc – przetwarzanie obrazów, operacje na obrazach (na przykład rozmazanie, operacje morfologiczne, geometryczne przekształcanie obrazów, wszelkiego typu filtry, wyrównywanie histogramu, analiza strukturalna).
- highgui – wysokopoziomowe funkcje wspomagające pobieranie obrazów z dysku, kamery, zapisywanie danych, wyświetlanie okien, interakcja z użytkownikiem, i tym podobne.
- video – analiza wideo, przepływ optyczny, usuwanie tła, i tym podobne.
- calib3d – kalibracja i obsługa wielu kamer, stereoskopia, wyszukiwanie map głębi, i tym podobne.
- features2d – klasyfikatory, detektory deskryptorów obiektów, wyszukiwanie wzorców, i tym podobne.

- objectdetect – klasyfikator kaskadowy, detekcja obiektów, i tym podobne.
- ml – uczenie maszynowe, klasyfikator Bayesa, klasyfikator KNN, drzewa decyzyjne i tym podobne.
- flann – Fast Approximate Nearest Neighbor Search – algorytm szybkiej aproksymaty wyszukiwania najbliższych sąsiadów.
- gpu – moduł do obsługi procesora graficznego.
- photo – odmalowywanie obszarów obrazów.
- stitching – wyszukiwanie i dopasowywanie punktów obrazów.
- nonfree – płatne algorytmy SIFT i SURF.

C.2.3 Zastosowanie

Biblioteka OpenCV jest stosowana głównie w grafice komputerowej do rozpoznawania i przetwarzania obrazów. Biblioteka jest używana również w robotyce i przemyśle. Używanie jej pozwala na skupieniu się na rozwiązywaniu wysokopoziomowych problemów, a nie na implementacji znanych algorytmów. Tam gdzie to możliwe parametry do wszystkich metod zostały dobrane optymalnie pod większość rozwiązań jednak zawsze możliwa jest ich manipulacja i dopasowanie do specyficznych zastosowań.

C.2.4 Przykładowy kod

Wczytanie obrazka

```
/** includes and namespaces */
int main() {
    Mat image = imread(„plik.jpg”, CV_LOAD_IMAGE_COLOR);
    if(image.data) {
        imshow(„Obrazek”, image);
    }
    waitKey(0);
}
```

Powyżej przedstawiono kompletny kod (poza załączeniami plików nagłówkowych i przestrzeniami nazw) realizujący wczytanie z dysku, a następnie wyświetlenie obrazka w oknie. Przykład ilustruje łatwość tworzenia kompletnego kodu dzięki zastosowaniu bardzo wysokopoziomowych funkcji biblioteki OpenCV.

Wyswietlenie obrazu z kamery

```
/** includes and namespaces */
int main() {
    Mat frame;
    VideoCapture capture(CV_CAP_ANY);
    while( waitKey(10) != 27 ) {
        capture >> frame;
        imshow( "Obrazek", image );
    }
    return 0;
```

Powyżej przedstawiono również kompletny kod programu wyświetlającego obrazy pobrane na bieżąco z zainstalowanej kamery w systemie.

Bardzo podobnie można uzyskać obrazy z materiału wideo przygotowanego wcześniej. Przykłady te ilustrują jak w prosty sposób można osiągnąć funkcjonalność, która przy zastosowaniu innych rozwiązań wymagałaby użycia wyrafinowanego kodu często kilkukrotnie przewyższającego swoją długością zaprezentowane przykłady.

Użycie procesora graficznego

```
gpu::GpuMat gpuImg(frame);
gpu::GpuMat gpuGrayImg;
gpu::cvtColor(gpuImg, gpuGrayImg, CV_BGR2GRAY);
gpu::dilate(gpuGrayImg, gpuGrayImg,
```

Kolejny przykład ilustruje użycie funkcji realizowanych na procesorze graficznym. W kodzie założono istnienie pobranego obrazka (w zmiennej frame).

Kod ten dokonuje konwersji koloru obrazka z kolorowego na szary a następnie wykonywana jest operacja dylatacji.

C.3 NVIDIA® CUDA™

C.3.1 Główne cechy

CUDA™, czyli opracowana przez firmę technologia wspierająca obliczenia równoległe przy użyciu procesora graficznego (GPU). W przeciwieństwie do bibliotek typu DirectX czy OpenGL, biblioteka CUDA™ oferuje wsparcie dla zaawansowanych obliczeń niekoniecznie związanych z grafiką. Biblioteka ta jest obsługiwana tylko przez karty graficzne producenta NVIDIA® jednak, mimo to zyskała ogromną popularność dzięki mnogości

urządzeń, na jakich możliwe jest jej działanie. Obecnie wszystkie nowe karty graficzne producenta oferują takie wsparcie i na obecną chwilę (rok 2012) zostało sprzedane ponad 128 milionów układów GPU z obsługą CUDA™ [2].

Biblioteka firmy NVIDIA® oferuje wsparcie dla rozwiązań astrofizycznych, biologicznych, chemicznych, z zakresu dynamiki płynów, elektromagnetycznych, sejsmicznych i wiele innych. Między innymi CUDA™ oferuje wsparcie dla przetwarzania wideo oraz renderowania grafiki.

C.3.2 Technologia

Rozwiązanie CUDA™ zostało wprowadzone w 2007 roku i obecnie dostępna jest już wersja 4.1. Biblioteka jest rozpowszechniona i chętnie używana przez tysiące programistów. Głównym językiem programowania dla którego została zaprojektowana to C, C++, ale istnieje wiele rozszerzeń umożliwiających używanie biblioteki programując w językach Fortran, Java, Python, .NET i wielu innych. Podobnie jak OpenCV, CUDA™ jest wieloplatformowa (Linux, MacOS, Windows).

CUDA™ implementuje dwie bardzo znane biblioteki obliczeniowe BLAS i FFT odpowiednio cuBLAS i cuFFT, jak również bibliotekę CUDPP (Data Parallel Primitives) oferujących algorytmy takie jak równoległą redukcję czy sortowanie.

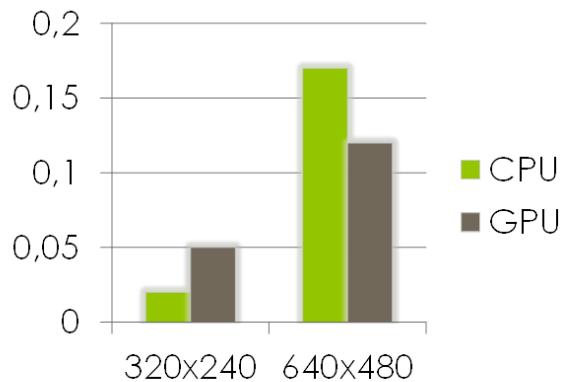
Do działania technologii CUDA™ wymagany jest osobny kompilator nvcc. Jest to kompilator potrafiący skompilować kod C jak i dyrektywy procesora graficznego. Kompilator ten posiada zdolność komplikacji do kodu C (wykonawanego na CPU) jak również obiektów wykonywanych na GPU. Pliki wykonywalne wymagają biblioteki cuda oraz biblioteki uruchomieniowej cudart.

C.3.3 Interfejsy programistyczne

CUDA™ oferuje dwa różne podejścia do tworzenia aplikacji wykorzystującej tę technologię. Pierwsze z nich to interfejs poziomu urządzenia. Programista tworzy kod wykonywany bezpośrednio na procesorze graficznym za pomocą DirectX Compute, OpenCL lub CUDA™ Driver API.

Drugim podejściem jest interfejs pozwalający na integracje. Programista programuje w C i zaznacza tylko pewne funkcje, które mają zostać wykonane na GPU zamiast CPU.

Oprócz dwóch głównych wymienionych powyżej warto zaznaczyć, że możliwe jest tworzenie aplikacji w innych językach (Fortran, MATLAB), jednak kod ten wywołuje funkcje biblioteki, a nie jest bezpośrednio komplikowany dla procesora graficznego.



Wykres C.1. Przyspieszenie działania aplikacji dzięki zastosowaniu technologii CUDATM (Oś pionowa czas w sekundach, oś pozioma rozdzielcość).

C.4 CUDATM wraz z OpenCV

Połączenie dwóch, opisanych powyżej technologii umożliwia stworzenie programu, który będzie potrafił przetwarzać grafikę w zaawansowany sposób jednocześnie wykorzystując potencjał kart graficznych umożliwiających wielokrotne przyspieszenie niektórych algorytmów. Dzięki takiemu połączeniu możliwe staje się zaawansowane przetwarzanie wideo w czasie rzeczywistym.

OpenCV wspiera tworzenie aplikacji wykorzystujących CUDATM poprzez łączenie struktur danych używanych przez oba rozwiązania. Dodatkowo w bibliotece OpenCV zaprogramowane zostały funkcje wykorzystujące CUDATM. Dzięki temu wykorzystanie procesora graficznego w bibliotece OpenCV jest możliwe nawet bez znajomości technologii CUDATM. W przypadku, gdy wymagana jest większa elastyczność rozwiązania, jak również gdy problem jest nietrywialny istnieje możliwość stworzenia aplikacji, której część jest kompilowana jako zwykły C++ i przetwarzana przez bibliotekę OpenCV, a część kompilowana jest przez kompilator nvcc i wykonywana dzięki technologii CUDATM.

Na wykresie C.1 pokazano możliwość przyspieszenia aplikacji z wykorzystaniem technologii CUDATM. Przykładowa aplikacja służy do rozpoznawania gestów i ruchu dloni z obrazu kamery internetowej. Brano pod uwagę tylko czas potrzebny na obliczenia (pominięto wyświetlanie wyników oraz pobranie obrazu z kamery).

C.4.1 Kod źródłowy

Dla pokazania możliwości połączenia CUDATM i OpenCV został stworzony prosty kod źródłowy pokazujący jak, w najprostszy sposób osiągnąć możliwość wykonywania wła-

snego kodu na procesorze graficznym w prosty sposób.

Poniższy kod nie jest kompletny, ma on na celu przedstawić jedynie fragmenty kodu wymagane do połączenia OpenCV z kodem procesora graficznego.

```
struct MYAPI_API Pixel{
    unsigned char b;
    unsigned char g;
    unsigned char r;
};

/* ... */

GpuMat gpuSrc, gpuDst;
mojaFunkcjaCUDA (
    (cv::gpu::DevMem2D<Pixel>) gpuSrc,
    (cv::gpu::DevMem2D<Pixel>) gpuDst );
```

Powyżej została przedstawiona część po stronie OpenCV. Na początku kodu tworzona jest prosta struktura danych przechowująca jeden punkt na obrazie. Następnie założono, że zmienna *gpuSrc* jest to obrazek źródłowy wypełniony danymi do edycji, a *gpuDst* pamięć, gdzie zapisany zostanie wynik.

```
static void mojaFunkcjaCUDA(
    const DevMem2D<Pixel>& src,
    DevMem2D<Pixel>& dst){
    dim3 block(16, 16);
    int grida = src.cols / block.x + !(src.cols % block.x);
    int gridb = src.rows / block.y + !(src.rows % block.y);
    dim3 grid(grida, gridb);
    gpuSimpleFunction<<<grid, block>>>(src, dst);
}
```

W pierwszych liniach kodu determinowana jest liczba wątków uruchamianych na procesorze graficznym w zależności od rozmiaru obrazka. Gdy wielkość obrazka jest stała można przyjąć tu stałe wartości i wtedy powyższy kod redukuje się do jednej linijki.

Kolejny kod przedstawia już instrukcje wykonywane po stronie procesora graficznego.

```
--global__ void gpuSimpleFunction(
    const DevMem2D<Pixel> src, DevMem2D<Pixel> dst){
    const int x = blockDim.x * blockIdx.x + threadIdx.x;
    const int y = blockDim.y * blockIdx.y + threadIdx.y;
```

```
if (x < src.cols && y < src.rows){  
    //odwrócenie kolorów  
    dst.ptr(y)[x] = 255 - src.ptr(y)[x];  
}  
}
```

Powyższy kod źródłowy przedstawia instrukcje wykonywane całkowicie po stronie procesora graficznego. Na początku determinowany jest aktualnie przetwarzany piksel, a następnie do pamięci wyjściowej przepisywana jest obliczona wartość nowego piksela. W podanym przykładzie jest to odwrotność wartości wejściowej (odwrócenie kolorów).

Powyżej zaprezentowane rozwiązanie ukazuje, że połączenie CUDA™ i OpenCV pozwala na osiągnięcie zamierzonego efektu niewielkim nakładem pracy.

C.5 Inne rozwiązania

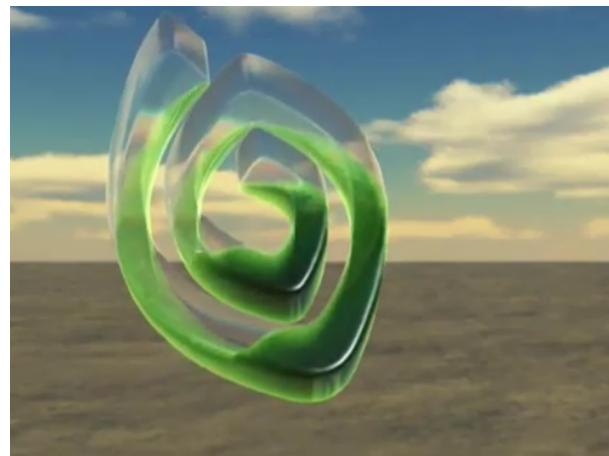
Istnieje wiele rozwiązań pozwalających na zrównoleglenie kodu i wykorzystanie potencjału kart graficznych. Inne rozwiązania to na przykład:

- OpenCL (Open Computing Library) – technologia wspierająca tworzenie programów na heterogenicznych platformach (na przykład CPU + GPU)
- Intel® MKL – obliczenia matematyczne równoległe na CPU (Brak wsparcia procesora graficznego)
- OpenMP (Open Multi-Processing – tylko CPU)
- Ati Stream – rozwiązanie podobne do CUDA™ lecz dla kart AMD/ATI

C.6 Podsumowanie

W pracy zostały omówione biblioteka programistyczne OpenCV jak również NVIDIA® CUDA™. Technologie te są przydane przy tworzeniu programów opartych na najnowszych osiągnięciach techniki w sposób łatwy i efektywny. Przedstawione zostały przykładowe kody źródłowe programów, dzięki czemu możliwe jest przeanalizowanie i wdrożenie rozwiązań we własnych projektach.

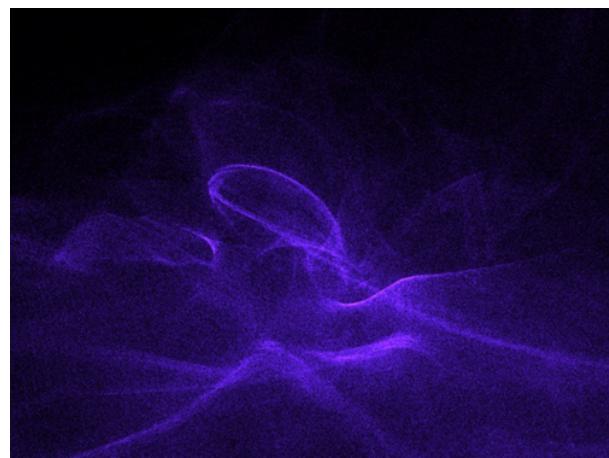
Na zrzutach ekranu C.2-C.5 przedstawione zostały programy wykorzystujące technologię CUDA™ oraz OpenCV.



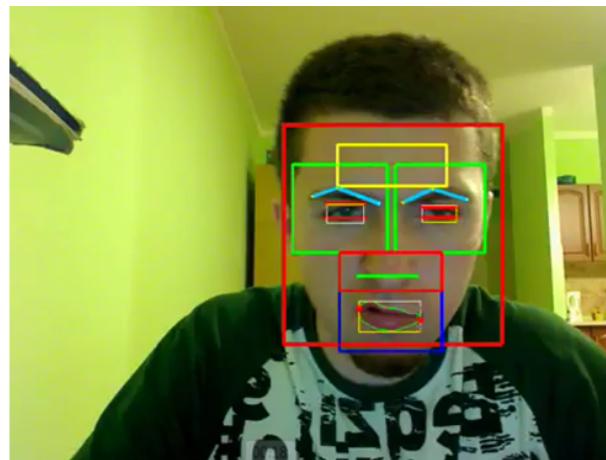
Zrzut z ekranu C.2. Program ilustrujący ciecz zamkniętą w przezroczystym logo firmy NVIDIA® – technologia CUDA™.



Zrzut z ekranu C.3. Wizualizacja 1 200 000 cząsteczek z oddziaływaniem dynamicznej siły w czasie rzeczywistym - technologia CUDA™.



Zrzut z ekranu C.4. Wykrywanie krawędzi – technologia OpenCV.



Zrzut z ekranu C.5. Wykrywanie deskryptorów twarzy (inaczej wykrywanie nastroju) - technologia OpenCV.

C.7 Bibliografia

- <http://opencv.willowgarage.com/wiki/>
- <http://developer.nvidia.com/category/zone/cuda-zone>.
- NVIDIA® CUDA™ Architecture ver 1.1. 2009 – pdf
- <http://www.khronos.org/opencl/>
- <http://openmp.org/wp/>
- <http://www.amd.com/stream>