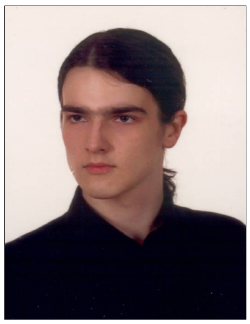
POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRYCZNY
INSTYTUT STEROWANIA I ELEKTRONIKI PRZEMYSŁOWEJ

PRACA DYPLOMOWA INŻYNIERSKA
na kierunku INFORMATYKA
specjalność: INŻYNIERIA OPROGRAMOWANIA



Krzysztof SZAWŁOWSKI

Nr albumu: 221211

Rok akad.: 2011/2012
Warszawa, 25.06.2011

WIELOWĄTKOWE PRZETWARZANIE OBRAZÓW Z WYKORZYSTANIEM BIBLIOTEK OPENCV ORAZ INTEL TBB

Zakres pracy:

1. *Wprowadzenie i sformułowanie celu pracy.*
2. *Programowanie wielowątkowe z wykorzystaniem biblioteki Intel TBB.*
3. *Architektura biblioteki OpenCV.*
4. *Analiza porównawcza wydajności przetwarzania jedno- i wielowątkowego na wybranych algorytmach przetwarzania obrazów.*
5. *Podsumowanie i wnioski.*

Kierujący pracą: dr inż. Witold Czajewski

Konsultant:

Termin złożenia pracy: 27.01.2011

Warszawa, dnia 27 stycznia 2012r.

Politechnika Warszawska
Wydział Elektryczny

OŚWIADCZENIE

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa inżynierska pt. Wielowątkowe przetwarzanie obrazów z wykorzystanie biblioteki OpenCV oraz Intel TBB:

- została napisana przeze mnie samodzielnie,
- nie narusza niczyich praw autorskich,
- nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam, że przedłożona do obrony praca dyplomowa nie była wcześniej podstawą postępowania związanego z uzyskaniem dyplomu lub tytułu zawodowego w uczelni wyższej. Jestem świadom, że praca zawiera również rezultaty stanowiące własności intelektualne Politechniki Warszawskiej, które nie mogą być udostępniane innym osobom i instytucjom bez zgody Władz Wydziału Elektrycznego.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Krzysztof Szawłowski.....

Wielowątkowe przetwarzanie obrazów z wykorzystaniem biblioteki OpenCV oraz Intel TBB

Streszczenie

Niniejsza praca inżynierska miała na celu przetestowanie istniejących rozwiązań z dziedziny programowania równoległego i współbieżnego zaimplementowanych w bibliotece OpenCV. Wielowątkowość wbudowana w bibliotekę została napisana przy pomocy Intel TBB. Rozdział pierwszy jest wprowadzaniem do pracy, zawarte w nim są informacje ogólne wprowadzające w tematykę pracy. Wymienione tutaj zostały wszystkie cele, które ustalone były przed rozpoczęciem wykonywania pracy inżynierskiej.

Prace nad projektem wymagały zaznajomienia się z obydwiema bibliotekami. Wymagane było też bliższe zapoznanie się z zagadnieniami programowania wielowątkowego. Z całą pewnością różni się ono od programowania sekwencyjnego. Występują w nim problemy, z którymi w programowaniu jednowątkowym nie trzeba się zmagać. Rozdział drugi i trzeci poświęcone są więc opisom bibliotek OpenCV oraz Intel TBB. Opisy w nich zawarte skupiają się tylko i wyłącznie na najważniejszy dla pracy aspektach obu bibliotek. W dziale drugim poza samą biblioteką Intel TBB napisane zostało krótkie wprowadzenie do programowania wielowątkowego.

W trakcie tworzenia pracy powstały dwa programy. Pierwszy z nich testuje funkcje wbudowane w bibliotekę OpenCV, które mają zaimplementowaną obsługę mechanizmów wielowątkowych przy pomocy biblioteki Intel TBB. Program ten został napisany w sposób obiektowy, więc można go łatwo rozbudować. Drugi program skupiał się na testach własnych algorytmów napisanych z wykorzystaniem wielowątkowości. Program ten także został napisany z zastosowaniem podejścia obiektowego. Czwarty rozdział to opis funkcjonalny oraz implementacyjny utworzonych programów. Przedstawia on szczegółowo, w jaki sposób działają oba programy, jakie produkują wyniki oraz w jaki sposób je obsługiwać. Ponadto każdy z programów został opisany od strony programowej.

Oba utworzone programy posłużyły do wykonania testów biblioteki OpenCV oraz Intel TBB. Wykonane testy pokazały, że zrównoleglenie wbudowane w OpenCV nie jest wystarczająco rozbudowane. Poza algorytmami, które mają zaimplementowaną wielowątkowość, dałoby radę zaimplementować ją także w innych algorytmach. Niemniej jednak, rozwój zrównoleglenia w bibliotece nie jest priorytetem jej autorów. Piąty rozdział to dokładny opis przeprowadzonych testów. Zebrane w nim są wyniki wszystkich testów. Wszystkie wyniki opatrzone są komentarzami i uwagami. Na końcu rozdziału

znajduje się ogólne zestawienie wszystkich testów, dzięki któremu w łatwy sposób można porównać jakość zrównoleglenia we wszystkich algorytmach.

Praca pokazuje także, że wykorzystywanie biblioteki Intel TBB we własnych algorytmach może przynieść duże zyski. Zyski te zależne są oczywiście, od rozwiązywanych problemów. Wydaje się jednak, że przy współczesnym kierunku rozwoju komputerów, programowanie równoległe będzie zyskiwało coraz większą wartość. Ostatni rozdział to wnioski z wykonanej pracy inżynierskiej. Podsumowana w nim jest cała wykonana praca. Podjęta tutaj również została próba odpowiedzenia na pytanie, czy zrównoleglenie wbudowane w bibliotekę OpenCV jest wystarczające. Ponadto w dziale tym przedstawione są propozycje dalszej rozbudowy pracy.

Multi-threaded processing of images with the use of OpenCV and Intel TBB libraries

Abstract

The aim of this BSc thesis was to test the existing solutions in the field of parallel and concurrent programming implemented in the OpenCV library. Multi-threading built-in into library was written using Intel TBB. The first chapter is an introduction to the thesis, it contains basic information about thesis subject. There is also a list of goals, which were necessary to accomplish the work on the thesis.

During the work on the project these two libraries had to be studied. It was also required to learn about issues of multi-threaded programming. It certainly differs from sequential programming. There are problems, with which in the single-threaded programming one does not have to struggle. The second and the third chapter are about OpenCV and Intel TBB libraries. They describe only the most important for the thesis parts of libraries. The second chapter contains also a small introduction to multi-threaded programming.

Two programs were written during the work on the thesis. The first of them tests mechanisms built into OpenCV library that support Intel TBB multi-threading. This program was written in object-oriented way, so it can be easily upgraded. The second program focuses on testing own multi-threaded algorithms. This program was also written in object-oriented way. The fourth chapter is an functional and implementational description of created programs. It gives detailed description about how these programs work, what is their output and how to use them.

Both programs were used to test OpenCV and Intel TBB libraries. Performed tests have shown that multi-threading built-in into OpenCV library is not very advanced. In addition to algorithms that are using multi-threading, it is possible to make other functions multi-threaded. However, the development of multi-threading built-in into the library is not a priority of it's authors. The fifth chapter is a detailed description of performed tests. Here are all the results of all the tests. The results ale described and commented. At the end of chapter, there is a little summary of all tests. It allows to easily compare quality of multi-threading in all algorithms.

The thesis shows also that using Intel TBB in own programs can bring performance boost. The boost depends of the algorithms. However, it seems that nowadays, parallel programming is gaining more and more importance. The last chapter contains summary of all the done work. There is an attempt to answer the question: Is the mult-ithreading build-in into OpenCV library advanced enough? It also contains suggestions for further work.

Spis treści

Podziękowania

Dziękuję bardzo serdecznie wszystkim, którzy wniesli wkład w moje wykształcenie. Szególnie dziękuję moim rodzicom, którzy wspomagali mnie w każdej chwili mojego życia, a także prowadzącemu pracę inżynierską dr inż. Witoldowi Czajewskiemu za doprowadzenie jej do końca oraz nakierowanie mnie na właściwe rozwiązania.

Krzysztof Szawłowski

Rozdział 1

Wstęp

Już od jakiegoś czasu można zauważyć, że komputery zmieniły kierunek rozwoju. Jeszcze parę lat temu procesory różnych producentów prześcigały się wzajemnie częstotliwościami taktowania, ilością pamięci podręcznej, czy przepustowością szyny FSB. Niestety, częstotliwość taktowania procesora osiągnęła już pewną granicę, której nie da się przekroczyć bez zmiany materiałów, z których zbudowane są procesory. Potrzeba produkowania coraz to wydajniejszych układów zmusiła firmy do poszukiwania innych sposobów podnoszenia wydajności swoich produktów.

Do dnia dzisiejszego rozwijano przeróżne technologie dla procesorów, umożliwiające im wykonywanie coraz to bardziej złożonych operacji w coraz krótszym czasie. Jednym ze sposobów było instalowanie w jednostce centralnej więcej niż jednego rdzenia, by wykonywane przez procesor instrukcje mogły być realizowane jednocześnie. Pomysł ten nie jest nowy, a wywodzi się głównie z idei podzielenia obliczeń na mniejsze porcje i wykonania ich na kilku maszynach jednocześnie. Istnieją też maszyny wieloprocesorowe, które również realizują ideę obliczeń równoległych. Jednak ze względu na dużą w dzisiejszych czasach popularność procesorów wielordzeniowych, to właśnie na nich skupia się opracowywany projekt inżynierski.

Obliczenia równoległe na jednym procesorze o wielu rdzeniach okupione są wieloma komplikacjami. Programu wykonującego się sekwencyjnie nie da się tak łatwo wykonać na kilku rdzeniach, gdyż bardzo często wynik następnych obliczeń wymaga ukończenia wcześniejszych. Procesor wykonując kolejne instrukcje nie może tak po prostu przerzucać pierwszej z nich na jeden rdzeń, a drugiej na drugi rdzeń. Przykładowo, przy wykonywaniu obliczeń matematycznej formuły 2+2*2 pierwszą operacją jest mnożenie, drugą zaś dodawanie. Operacji tych nie można zamienić. Mało tego, wykonując drugą z nich musimy znać wynik pierwszej. Dokładnie tak samo jest w przypadku działania instrukcji procesora.

Istnieje jednak wiele instrukcji, które mogą być liczone niezależnie od siebie. Najprostszym przykładem jest dodawanie do siebie dwóch wektorów. Każdy z rdzeni może dodawać dwa elementy, a wynik zapisywać w wektorze trzecim. W takim przypadku teoretycznie przy dwóch rdzeniach można by uzyskać dwukrotnie skrócony czas obliczeń. Zadania procesorów bardzo często opierają się na wykonywaniu tych samych obliczeń na różnych danych wejściowych. Przy odpowiednim zarządzaniu zasobami, zadania te mogą być wykonywane na kilku rdzeniach, a czas potrzebny do ich ukończenia może być znacznie skrócony.

W przetwarzaniu obrazów prawie zawsze mamy do czynienia z operacjami na ogromniej ilości pikseli. Stosując filtr dla obrazu, szukając najjaśniejszego piksela, dokonując różnych przekształceń, prawie zawsze operujemy na pojedynczych pikselach. Rozdzielczość obrazu może wahać się od kliku pikseli do kilku milionów lub miliardów pikseli. Jeśli mamy więc do czynienia z tak ogromną ilością obliczeń, obliczeń które można wykonywać niezależnie od siebie, warto wykorzystać możliwości współczesnych procesorów z dziedziny przetwarzania wielowątkowego.

Wielowątkowość zastała wykorzystana przez twórców biblioteki OpenCV. Pierwotnie funkcje tej biblioteki były pisanie z myślą o wykorzystaniu biblioteki OpenMP. Jednak od pewnego czasu wszelkie funkcje wykorzystujące obliczenia równoległe pracują w oparciu o inną bibliotekę, a mianowicie Intel TBB. Połączenie obu bibliotek ma na celu wykorzystanie potencjału współczesnych procesorów przy przetwarzaniu obrazów.

1.1 Cel i układ pracy

Zasadniczym celem pracy jest sprawdzenie, czy zrównoleglenie istniejące w bibliotece OpenCV jest wystarczające, czy może dałoby się zrealizować je lepiej lub inaczej. W czasie realizacji pracy wykonane zostaną testy mające na celu sprawdzenie jak sprawuje się sama biblioteka Intel TBB oraz czy warto stosować ją pisząc własny kod.

Zadaniem pracy jest analiza istniejących rozwiązań zrównoleglających w bibliotece OpenCV, uwzględniając głównie zysk jaki niosą ze sobą owe rozwiązania. Ponadto praca zakłada zaproponowanie własnych rozwiązań zrównoleglających obliczenia w funkcjach biblioteki i porównanie ich z tymi wbudowanymi. Realizacja części praktycznej opiera się na opracowaniu oraz zaimplementowaniu programu umożliwiającego przetestowanie zysku z zastosowania wielowątkowości w bibliotece OpenCV, a także napisania i przetestowania własnych algorytmów wielowątkowych wykorzystujących bibliotekę Intel TBB.

Realizacja całej pracy wymaga osiągnięcia mniejszych celów:

1. przygotowanie środowiska pracy, stworzenie projektu w MS Visual Studio umożliwiającego kompilację programu w dwóch wersjach; pierwsza z nich ma wykorzystywać bibliotekę OpenCV wykorzystującą Intel TBB, druga z nich ma używać OpenCV bez Intel TBB
2. odnalezienie funkcji w bibliotece OpenCV, które mają zaimplementowane wstawki zrównoleglające
3. napisanie testów dla odnalezionych funkcji, funkcje te wymagają przygotowania odpowiednich danych, wczytania obrazków i odczytania wyników
4. przygotowanie mechanizmu umożliwiającego dokładny pomiar czasu
5. wykonanie napisanych wcześniej testów wraz z pomiarem czasu wykonywania
6. napisanie własnych algorytmów wykorzystujących Intel TBB wraz z OpenCV oraz algorytmów wykorzystujących tylko Intel TBB
7. wielokrotne wykonanie pomiarów zysku ze zrównoleglenia, w celu uśrednienia wyników i wyeliminowania błędów pomiarowych
8. wyciągnięcie wniosków z otrzymanych wyników, porównanie istniejących rozwiązań z własnymi algorytmami
9. próba odpowiedzenia na pytanie: czy zrównoleglenie zawarte w bibliotece OpenCV jest wystarczające?

Pomijając wstęp, praca podzielona jest na pięć rozdziałów. Rozdział pierwszy poświęcony jest zagadnieniom programowania równoległego i zawarty w nim jest krótki opis biblioteki Intel TBB. Opis ten dotyczy głównie sposobu zarządzania pracą wątków w bibliotece, a także przedstawia jej podstawowe funkcje i struktury danych.

Rozdział drugi poświęcony jest bibliotece OpenCV. Wyjaśnione w nim zostało czym jest ta biblioteka, do czego służy i w jaki sposób podjęto w niej próby zrównoleglenia. Wypisane tu zostały funkcje, w których autorzy podjęli próbę zrównoleglania obliczeń.

Trzeci rozdział to szczegółowy opis funkcjonalny i implementacyjny programów, które powstały w czasie pisania pracy inżynierskiej. Rozdział ten opisuje także kompilację biblioteki OpenCV wykorzystującej Intel TBB.

W rozdziale czwartym opisane są testy przeprowadzone w trakcie reazlizacji projektu. Otrzymane wyniki przedstawione są w postaci tabel, a każda z tabel opatrzona jest komentarzem.

Ostatni rozdział zawiera wnioski z przeprowadzonej analizy. Opisane w nim są efekty i konkluzje wynikające z przeprowadzonych badań.

Rozdział 2

Programowanie wielowątkowe z wykorzystaniem biblioteki Intel TBB

Intel TBB (Theading Buildding Blocks) - bibliteka została wprowadzona przez Intela 26 sierpnia 2006 roku. Przeznaczona jest do tworzenia oprogramowania wykorzystującego procesory wielordzeniowe. Upraszcza ona pisanie programów równoległych poprzez zwolnienie użytkownika z konieczności zarządzania wątkami oraz przydzielania im zadań. Biblioteka ta pozwala w prosty sposób uniknąć często trudnych do obejścia problemów zakleszczania się wątków. Umożliwia także łatwiejsze korzystanie z tych samych obszarów pamięci przez różne wątki, dzięki prostemu mechanizmowi zamków (mutex). Biblioteka traktuje operacje jako zadania i zarządza nimi. Dzięki wbudowanym w bibliotekę mechanizmom, zadania rozkładane są równomiernie na wszystkie rdzenie, a pamięć cache wykorzystywana jest w sposób efektywny.

W bibliotece wykorzystywany jest tzw. *task stealing*, czyli podkradanie zadań. Na początku wszystkie rdzenie dostają taką samą liczbę zadań do wykonania. Zadania te jednak nie są wykonywane zupełnie równolegle, dlatego często niektóre rdzenie kończą swoją pracę wcześniej niż inne. W takich właśnie przypadkach biblioteka dynamicznie przydziala nowe zadania rdzeniom, które kończą obliczenia szybciej. Zadania te są brane z puli zadań innych rdzeni które działają wolniej. Pozwala to w efektywny sposób wykorzystywać pełen potencjał procesorów wielordzeniowych.

Dostępność biblioteki jest dosyć szeroka. Mimo, że wprowadziła ją firma Intel, to działa ona także z wielordzeniowymi procesorami innych firm. Aktualnie dostępna jest na systemy operacyjne Windows, Linux oraz Mac OS. Bibioteka realizuje założenia uniezależniania programisty od sprzętu. Programy pisany z wykorzystaniem tej biblioteki będą działał na różnych maszynach z

różnymi procesorami.

2.1 Pomiar przyspieszenia

Przyspieszenie jest miarą, która określa zysk z zastosowania mechanizmów wielowątkowych. Współczynnik ten można obliczyć ze wzoru 2.1.

$$S(n,p) = \frac{T(n,1)}{T(n,p)} \tag{2.1}$$

gdzie :
S(n,p) - przyspieszenie
T(n,p) - czas wykonania zadania n na p procesorach
n - wielkość zadania
p - liczba procesorów/rdzeni

Zysk ze zrównoleglenia nie może być większy niż liczba procesorów, dlatego zachodzi nierówność: 2.2

$$S(n,p) \leqslant p \tag{2.2}$$

Każdy program działający współbieżnie składa się z dwóch części. Pierwsza z nich musi być wykonywana sekwencyjnie, druga to ta, którą da się zrównoleglić. Zysk ze zrównoleglenia zależny jest od udziału każdej z tych części w programie, a także od narzutu komunikacyjnego pomiędzy wątkami.

Jeżeli oznaczymy udział sekwencyjny jako $\beta(n)$ i założymy, że część sekwencyjną da się idealnie zrównoleglić na nieskończoną liczba procesorów, to prawdziwy jest wzór 2.3

$$S(n,p) = \frac{1}{\beta(n) + \frac{1-\beta(n)}{p}} \tag{2.3}$$

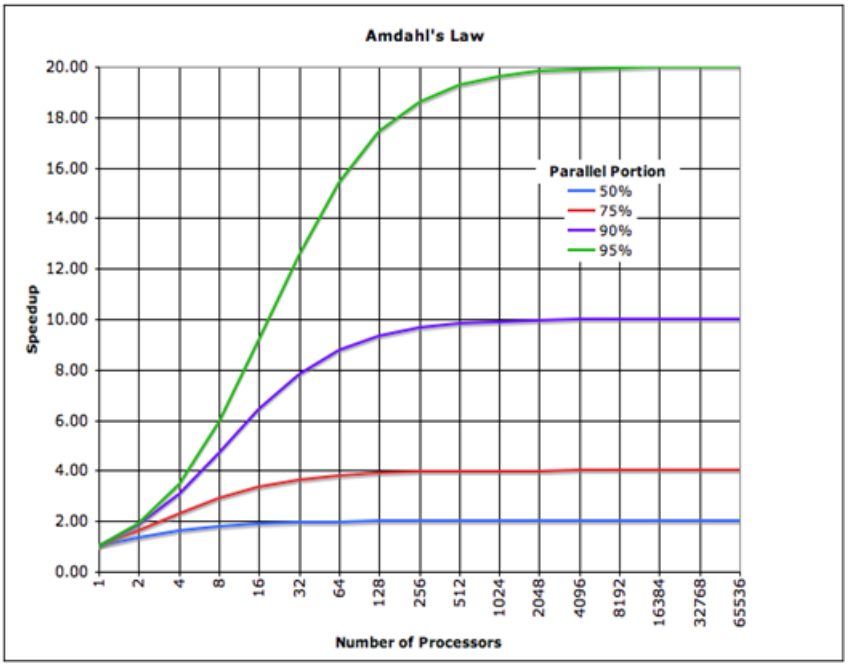
z którego wynika, że wartość przyspieszenia spada wraz ze wzrostem udziału częsci sekwencyjnej w programie. O przedstawionym fakcie mówi prawo Ahmdala.

2.1.1 Prawo Ahmdala

Prawo to brzmi: „Nawet przy użyciu dowolnie wielu procesorów, obliczeń nie da się przyspieszyć bardziej, niż wynosi odwrotność udziału części sekwencyjnej w programie wykonywanym na jednym procesorze" [9]. To właśnie część sekwencyjna najbardziej decyduje o tym, czy dany program da się dobrze zrównoleglić, czy też nie. Prawo to zostało zapisane wzorem 2.4

$$S(n,p) \xrightarrow[n\to\infty]{} \frac{1}{\beta(n)} \tag{2.4}$$

Kiedy rośnie liczba procesorów, maleje udział części równoległej w programie. Niestety, udział części sekwencyjnej cały czas pozostaje taki sam. Program musi wykonać ją w danym czasie i nie da się jej w żaden sposób skrócić. Nakłady komunikacyjne są najczęściej mniej znaczące niż udział części sekwencyjnej, dlatego zostaną tu pominięte. Warto jednak o nich pamiętać i zdawać sobie sprawę, że wpływają one w pewnym stopniu na współczynnik przyspieszenia. Przykładowe zależności współczynnika przyspieszenia od liczby procesorów widoczne są na rysunku 2.1.



Rysunek 2.1: Wykres zależnośći współczynnika przyspieszenia w zależności od liczby procesorów. Zaczerpnięte z [8].

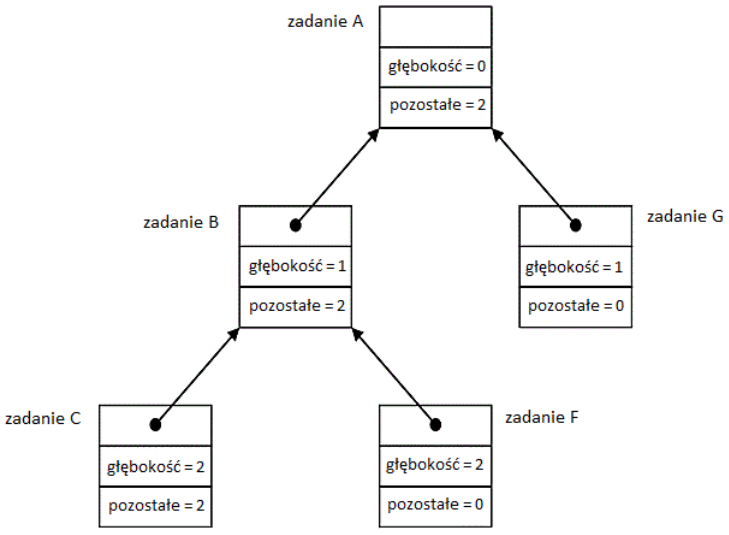
Niebieska linia obrazuje program, gdzie udział części sekwencyjnej jest największy, podczas gdy linia zielona to program, gdzie udział ten jest najmniejszy. Jak widać, użycie coraz większej liczby procesorów nie przynosi liniowego wzrostu przyspieszenia. Kiedy udział części nie dającej się zrównoleglić wynosi 50% to przyspieszenie nie osiągnie wartości 2. Z kolei 5-cio procentowy udział części sekwencyjnej przy ponad sześćdziesięciu pięciu tysiącach procesorów nie pozwoli osiągnąć dwudziestokrotnego przyspieszenia.

Dla każdego programu istnieje więc linia opłacalności, która określa czy warto dokładać jeszcze procesorów, by uzyskać większe przyspieszenie.

2.2 Schemat działania zarządcy zadań w Intel TBB

Zarządca zadań w bibliotece Intel TBB wzorowany był na zarządcy Cilk (źródło: [6]). Oba systemy zarządzania stosują podkradanie zadań. Według założeń, każdy wątek ma własną pulę zadań do wykonania. Podejście takie eliminuje konkurencję wątków o zadania, która ma miejsce w przypadku globalnej puli zadań. Pozawala także na tworzenie dodatkowych lokalnych zadań w czasie działania programu. Kiedy pula zadań dla jednego z procesów kończy się, wątek próbuje podkraść zadania losowo wybranego innego wątku. Właśnie takie podejście odróżnia zarządce zadań z biblioteki Intel TBB od innych.

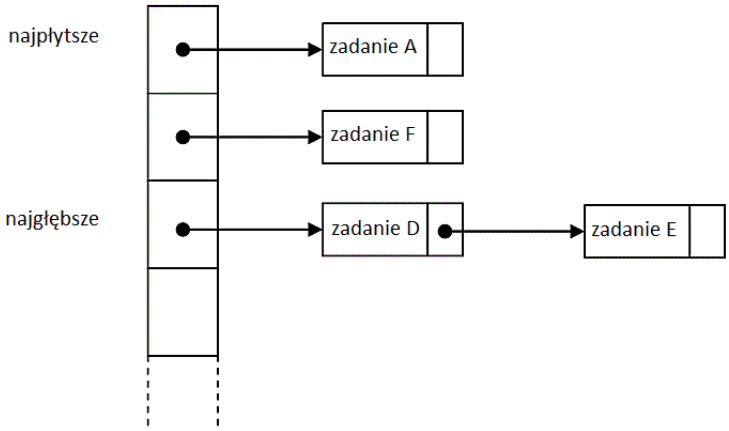
Pula zadań dla każdego z wątków przetrzymywana jest najczęściej w postaci drzewa. Rysunek 2.2 obrazuje wygląd puli wszystkich zadań do wykonania przez dany wątek.



Rysunek 2.2: Schemat drzewa zadań. Na podstawie [6].

Każde zadanie jest węzłem w drzewie. Węzeł przetrzymuje informacje o tym jak głęboko w drzewie znajduje się on sam oraz o tym ilu jego potomków nie skończyło jeszcze pracy. Zadania z puli wszystkich zadań przenoszone są

do puli zadań możliwych do wykonania w danej chwili. Pula ta jest przetrzymywana w formie wektora list zadań. Schemat takiego wektora znajduje na rysunku 2.3.



Rysunek 2.3: Schemat list zadań. Na podstawie [6].

Do wektora trafiają tylko te zadania, które w danej chwili mogą być wykonane, tzn takie, gdzie licznik pracujących potomków równy jest zeru. Kolejność zadań w wektorze ściśle odwzorowuje ich głębokość w drzewie. Na początku znajdują się zadania, które mają najmniejszą wartość indeksu głębokości. Na końcu są zadania znajdujące się najgłębiej w drzewie.

Intel TBB stara się utrzymać pracę wszystkich rdzeni na raz, najdłużej jak to tylko możliwe. Podstawową strategią działania algorytmu zarządcy TBB jest motto „kradnij z początku, wykonuj z końca ". Oznacza to, nie mniej tylko tyle, że wykonywanie zadań odbywa się od ostatniego elementu listy, a podkradane są zadania z początku listy. Podejście takie jest słuszne, gdyż zadania odkładane na stosie są w kolejności od pierwszego do ostatniego. Ostatnie zadanie z listy znajduje się więc na szczycie stosu. Żeby się do nich dostać nie trzeba przerzucać całego stosu. Kiedy jeden proces skończy wykonywać wszystkie swoje zadania, przejmuje zadania znajdujące się na spodzie stosu innego wątku, zmniejszając go w ten sposób.

Poza standardowym działaniem zarządcy, programista może użyć także innych sposobów zarządzania. W niektórych przypadkach są one bardziej efektywne niż przedstawiona powyżej metoda. Są to jednak przypadki szczególne. Jednym z nich jest stan, w którym drzewo staje się naprawdę duże i powoduje przepełnienie stosu. Intel TBB ma zaimplementowany mechanizm zadań łączonych. Kiedy drzewo staje się zbyt duże, zostaje ono podzielone na

mniejsze drzewa, a pamięć drzewa nadrzędnego zostaje zupełnie zwolniona. Dopiero, kiedy drzewa potomne skończą być wykonywane, następuje przywrócenie drzewa macierzystego, przyciętego o drzewa potomne. Zwiększa to co prawda nakład obliczeniowy, jednak w pewnych przypadkach umożliwia wykonanie zadań, gdzie stos mógłby ulec przepełnieniu.

2.3 Dostępne funkcje

Biblioteka Intel TBB wyposażona jest w gotowe funkcje oraz struktury danych upraszczające zrównoleglanie programów. Oto lista najczęściej używanych elementów:

- parallel_for - funkcja będąca odpowiednikiem pętli for, opisana jest dokładniej w dziale 2.3.1
- parallel_reduce - dokonuje redukcji na wszystkich elementach przedziału
- parallel_scan - wykonuje obliczenia równoległego prefixu
- parallel_while, parallel_do, parallel_do_while - odpowiedniki sekwencyjnych pętli
- concurrent_queue - implementacja kolejki umożliwiająca operowanie na niej wielu wątków
- concurrent_vector - implementacja wektora dająca możliwość wykonywania na nim wielu zadań na raz, bez naruszania jego kolejności
- concurrent_hash_map - bezpieczna pod względem przetwarzania wielowątkowego wersja hash mapy
- scalable_malloc, scalable_free, scalable_realloc - równoległe odpowiedniki podstawowych funkcji do zarządzania pamięcią
- mutex - podstawowy mechanizm zamków, umożliwiający synchronizację dostępu do pamięci itp, opisany szerzej w dziale 2.3.2
- spin_mutex, spin_rw_mutex ,queuing_rw_mutex, recursive_mutex - różne modele zamków

Poza wymienionymi istnieją jeszcze inne funkcje oraz struktury danych, praca nie przewiduje jednak aż tak złożonej analizy biblioteki. Więcej informacji na ten temat można znaleźć w [5].

2.3.1 Parallel_for

Funkcja służąca do wykonywania równoległych obliczeń w pętlach for. Jej implementacja zawarta jest w pliku nagłówkowym: tbb/parallel_for.h. Funkcję można wywołać na jeden z trzech sposobów:

```
Func parallel_for( Index first, Index_type last, const Func& f
                   [, task_group_context& group] );
```

W pierwszym przypadku podajemy trzy argumenty: indeks pierwszego elementu, indeks ostatniego elementu oraz wskaźnik na funkcję, która ma być wykonywana dla każdego elementu. Funkcja musi przyjmować tylko jeden parametr, który jest indeksem.

```
Func parallel_for( Index first, Index_type last,
                   Index step, const Func& f
                   [, task_group_context& group] );
```

Drugie wywołanie jest bardzo podobne do pierwszego. Dodatkowym parametrem funkcji wartość korku, którym algorytm ma się przemieszczać od wartości początkowej do końcowej.

```
void parallel_for( const Range& range, const Body& body,
                   [, partitioner[, task_group_context& group]] );
```

Trzecie wywołanie jest zupełnie inne. Przyjmuje dwa parametry: zasięg i ciało klasy. Zasięg jest strukturą, która posiada swój zakres i iterator. Istnieją różne rodzaje zasięgów: jedno-, dwu- lub trój-wymiarowe. Zasięg jednowymiarowy porównywalny jest do przedziału wartości dla zmiennej „i" w pętli „for" z określonym krokiem (np. „i++"). Parametr body to wskaźnik na klasę o następującej budowie:

```
class Body{

    Body::Body( const Body& )
    Body::~Body()
    void Body::operator()( Range& range ) const

}
```

Tworzona klasa musi zawierać konstruktor, destruktor oraz operację operator(), przyjmującą jeden parametr w postaci wskaźnika na obiekt klasy zasięg.

Poniżej znajduje się przykład użycia funkcji paraller_for zaczerpnięty z [5]. Przykładowy program liczy średnią z wektora liczb.

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"

using namespace tbb;

struct Average {
    const float* input;
    float* output;
    void operator()( const blocked_range<int>& range ) const {
        for( int i=range.begin(); i!=range.end(); ++i )
            output[i] = (input[i-1]+input[i]+input[i+1])*(1/3.f);
    }
};

// Note: Reads input[0..n] and writes output[1..n-1].
void ParallelAverage( float* output, const float* input, size_t n
) {
    Average avg;
    avg.input = input;
    avg.output = output;
    parallel_for( blocked_range<int>( 1, n-1 ), avg );
}
```

Na początku oczywiście znajdują się odpowiednie dyrektywy do kompilatora służące zaimportowaniu odpowiednich nagłówków:

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"
```

Później deklaracja używanej przestrzeni nazw, w tym przypadku przestrzeni nazw tbb:

```
using namespace tbb;
```

Następnie zadeklarowana zostaje struktura będąca realizacją klasy potrzebnej do wykonania funkcji parallel_for. Jak widać struktura ta posiada funkcję operator(), a jedynym jej parametrem jest wskaźnik na obiekt typu blocked_range.

```
struct Average {
    const float* input;
    float* output;
    void operator()( const blocked_range<int>& range ) const {
        for( int i=range.begin(); i!=range.end(); ++i )
            output[i] = (input[i-1]+input[i]+input[i+1])*(1/3.f);
    }
};
```

Na samym końcu znajduje się ciało funkcji wykorzystującej parallel_for. Równie dobrze kod w niej zawarty mógłby znajdować np. w głównej funkcji programu:

```
void ParallelAverage( float* output, const float* input, size_t n
) {
    Average avg;
    avg.input = input;
    avg.output = output;
    parallel_for( blocked_range<int>( 1, n-1 ), avg );
}
```

Wykorzystanie tylko tej jednej funkcji może wprowadzić do kodu przyspieszenie rzędu ilości rdzeni na procesorze. Wszystko zależy od problemu. Jeśli program składa się praktycznie tylko z samych pętli, które mogą być wykonywane niezależnie, to zastosowanie zrównolelgonej wersji pętli for da bardzo dobre rezultaty.

W celu podkreślenia różnic w sposobach wykorzystywania funkcji parallel_for, pokazany zostanie jeszcze jeden przykład. Tym razem sumowanie odbędzie się wykorzystując wywołanie funkcji parallel_for z trzema argumentami (pierwszy przedstawiony sposób wywoływania funkcji), gdzie pierwszy parametr to początek przedziału, drugi to jego koniec, a trzeci jest wskaźnikiem na funkcję przyjmującą jeden parametr w postaci liczby całkowitej (int). Poniżej znajduje się kod całego przykładu:

```
#include "tbb/parallel_for.h"

using namespace tbb;

float* out;
float* in;

void average(int i){
    out[i] = (in[i-1]+in[i]+in[i+1])*(1/3.f);
}

// Note: Reads input[0..n] and writes output[1..n-1].
void ParallelAverage( float* output, const float* input, size_t n
) {
    in = input;
    out = output;
    parallel_for( 1, n-1, &average );
}
```

Tak samo jak w poprzednim przypadku, na samym początku znajdują się dyrektywy dla kompilatora, umożliwiające wykorzystywanie funkcji zawartych w bibliotece TBB. Poza dyrektywami jest tutaj deklaracja używanej przestrzeni nazw. Następnie zadeklarowane są zmienne będące zmiennymi prywatnymi modułu bądź klasy:

```
float* out;
float* in;
```

Zmienne te są zadeklarowane w ciele klasy/modułu, gdyż muszą one być dostępne dla funkcji average(), która może przyjmować tylko jeden parametr. Kolejna jest deklaracji tejże właśnie funkcji:

```
void average(int i){
    out[i] = (in[i-1]+in[i]+in[i+1])*(1/3.f);
}
```

Będzie ona wykorzystywana przez zrównoleglоną wersję pętli for. Widać także, że przyjmuje ona tylko jeden parametr. Wartość przekazywanego parametru to indeks na element, który ma zostać policzony. Na końcu znajduje się deklaracja funkcji głównej:

```
void ParallelAverage( float* output, const float* input, size_t n
) {
    in = input;
    out = output;
    parallel_for( 1, n-1, &average );
}
```

Funkcja ta przypisuje zmiennym lokalnym otrzymane parametry, po czym uruchamia zrównoleglоną pętlę parallel_for. Zarządca zadań z biblioteki TBB zajmuje się rozdzieleniem wszystkich wywołań tej funkcji i przydziela je odpowiednim wątkom. W wywołaniu widać wskaźnik na zadeklarowaną wcześniej funkcję.

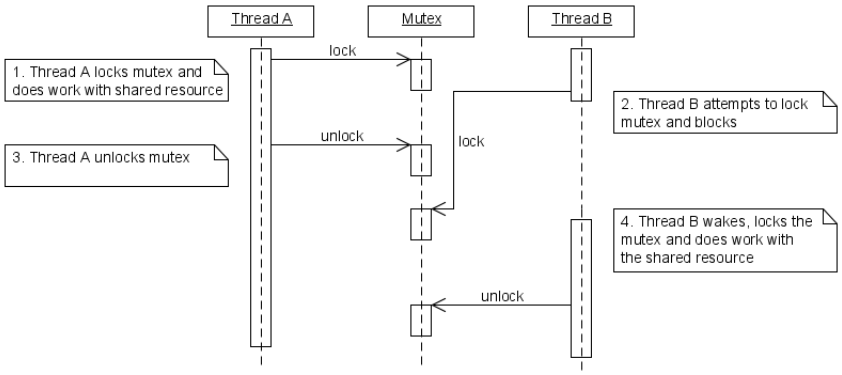
Drugi przykład wydaje się być bardziej przejrzysty, jednak tak naprawdę pierwszy z nich jest wydajniejszy i lepiej oddziela obliczenia równoległe od programu głównego. To wszystko za sprawą zarządcy zadań. W przypadku drugiego przykładu, każda pojedyncza iteracja po pętli for wykonywana jest na oddzielnym wątku i traktowana jest jako oddzielne zadanie. Podejście takie może stworzyć bardzo dużą liczbę zadań, w przypadku gdy ilość elementów w wektorze jest naprawdę duża. W pierwszym przykładzie, wektor zostanie automatycznie podzielony na bloki. Bloków tych będzie o wiele mniej niż elementów wektora. Każdy z wątków w pojedynczym zadaniu dostanie porcję danych, a nie pojedynczy element. Raz uruchomiony wątek będzie pracował dłużej, a narzut komunikacyjny zostanie zmniejszony. Dlatego właśnie, wywołanie funkcji parallel_for z pierwszego przykładu jest podejściem zalecanym przez twórców biblioteki Intel TBB.

2.3.2 tbb_mutex

Zamek (z ang. Mutex) - jest strukturą służącą synchronizacji dostępu wielu wątków do określonych zasobów w krytycznych częściach programu

działającego współbieżnie. Podczas pracy takiego programu zdarza się, że wątki muszą operować na jednej zmiennej, pisać do jednego pliku bądź przekształcać te same obszary pamięci. Działanie takie może doprowadzić do błędów, gdy np. jeden wątek inkrementuje pewną zmienną, a drugi w tym samym czasie ją dekrementuje. W celu uniknięcia takich problemów stosuje się różne mechanizmy. Jednym z nich jest właśnie mechanizm zamka.

W biliotece Intel TBB istnieje wiele różnych implementacji zamków. Najbardziej popularnym, a także jedynym wykorzystanym w tej pracy, jest mechanizm zamka prostego. Schemat działania zamków prostych widoczny jest na rysunku 2.4



Rysunek 2.4: Schemat działania zamków z [7]

Gdy jeden wątek dotrze do sekcji krytycznej sprawdza, czy zamek nie został już zamknięty. Jeśli nie, to sam go zamyka i rozpoczyna wykonywanie sekcji krytycznej. Po zakończeniu obliczeń wątek odblokowuje zamek i wykonuje swój kod dalej. Jeśli zamek w momencie dotarcia do części krytycznej jest zamknięty, to wątek musi poczekać na jego otwarcie. Dopiero po otworzeniu zamka, wątek może przystąpić do wykonywania części krytycznej. W przypadku, kiedy na otwarcie zamka czeka więcej niż jeden wątek, kolejność dostępu do zamka przydzielana jest według kolejności odwołań.

W bibliotece intel TBB zamek prosty zawarty jest w pliku nagłówkowym "tbb/mutex.h", a deklarujemy go jak każdą klasę czy zmienną w następujący sposób:

```
Mutex my_mutex;
```

Teraz przed wykonaniem sekcji krytycznej należy wykonać operację zamknięcia zamka:

```
my_mutex.lock();
```

Po zakończeniu obliczeń należy otworzyć zamek:

```
my_mutex.unlock();
```

Dopiero wtedy inne wątki będą mogły przystąpić do wykonywania sekcji krytycznej.

Przy używaniu zamków należy uważać. Jeśli będą się one pojawiały zbyt często i wszystkie wątki będą czekały na wykonanie pewnej sekcji, to może się okazać ze zysk ze zrównoleglenia jest bardzo mały lub zerowy. Mechanizm ten jest jednak niezbędny do prawidłowego działania niektórych programów. Programista musi być świadomy, że wątki mogą sobie nawzajem przeszkadzać lub uniemożliwiać prawidłową pracę. Mechanizmy synchronizacji są bardzo ważnym zagadnieniem programowania równoległego i współbieżnego.

Rozdział 3

Architektura biblioteki OpenCV

OpenCV jest biblioteką funkcji wykorzystywanych do obróbki obrazów. Zapoczątkowana została przez firmę Intel, jednak przekształcona została do projektu wolnego i jest nim do chwili obecnej. Biblioteka ta jest zbiorem ponad pięciuset funkcji z zakresu przetwarzania obrazów, wizji komputerowej i uczenia maszynowego(z ang. machine learning). Daje ona możliwość dowolnego przekształcania obrazów, wykrywania obiektów na obrazach, detekcji ruchu, a także konwersji przestrzeni barw, czy formatów zapisu obrazu. Praca skupiać się jednak będzie głównie na funkcjach, które mają zaimplementowane mechanizmy wykorzystujące wiele rdzeni procesorów. Warto dodać, że w bibliotekę zaimplementowana jest także możliwość wykorzystania kart graficznych przy pomocy technologi CUDA [4].

Biblioteka ta pierwotnie napisana została w języku C, jednak z powodu dużej popularności powstało do niej wiele różnych nakładek. Nakładki te umożliwiają wykorzystywanie funkcjonalności biblioteki w innych językach programowania i środowiskach. Do najważniejszych należą : C++, C#, Python oraz Matlab. Biblioteka dostępna jest na większość systemów operacyjnych m. in. Windowsa, Linuxa i Mac OS'a, a ostatnio można ją uruchomić nawet na platfrmach mobilnych takich jak IOS czy Android.

Od wersji 2.0 całkowitej przebudowie uległ interfejs biblioteki. Wcześniej praktycznie cała funkcjonalność znajdowała się w czterech modułach. Od wspomnianej wersji, biblioteka została rozbita na więcej modułów, a wywołania funkcji uległy zmianie. Co prawda, biblioteka jest kompatybilna wstecz, tzn. można wykorzystywać stary interfejs, jednak autorzy zalecają przystosowanie się do nowego nazewnictwa funkcji i struktur danych. Więcej o samej bibliotece można znaleźć tu [1]

3.1 Podział na moduły

Biblioteka podzielona została na moduły. Każdy z modułów jest zbiorem funkcji należących do określonej dziedziny. Wewnętrzna struktura biblioteki pozwala użytkownikowi na korzystanie tyko z tych jej fragmentów, których potrzebuje on w danej chwili. Nie trzeba zawierać w programie całej biblioteki. Jeśli program przewiduje użytkowanie tylko jednego modułu to można dodać tylko jeden plik nagłówkowy odpowiadający temu modułowi. Zmniejsza to ostateczny rozmiar programu i nie powoduje zbędnego nieporządku przy nazewnictwie funkcji. Moduły zawarte w bibliotece OpenCV to:

- core - główna funkcjonalność
- imgproc - przetwarzanie obrazów
- highgui - wysokopoziomowy interfejs użytkownika i obsługa wejścia-/wyjścia
- video - analizowanie video
- calib3d - kalibracja kamery i rekonstrukcja 3D
- features2d - przetwarzanie cech obrazów 2D
- objdetect - wykrywanie obiektów
- ml - uczenie maszynowe
- flann - grupowanie i poszukiwania wielowymiarowe
- gpu - wizja komputerowa wspomagana przez jednostki graficzne

Dokładniejszy opis modułów znajduje się w następnych podrozdziałach. Więcej informacji na temat budowy i wykorzystnie poszczególnych modułów można znaleźć w [2].

3.1.1 Core

Główny i najważniejszy moduł biblioteki. Zawiera budowę podstawowych struktur danych i funkcji wykorzystywanych przez pozostałe moduły. Umożliwia tworzenie tych struktur, wypełnianie ich danymi, operowanie an nich, ich przekształcanie oraz bezpieczne usuwanie. Moduł ten ujednolica sposób oraz formę przechowywania danych przez bibliotekę.

Przykładowymi strukturami danych są np. punkt, prostokąt, wektor, macierz czy wielowymiarowa struktura danych Mat. Mogą one być wykorzystywane przez inne modułu, np. imgproc, który posługuje się obrazami przetrzymywanymi w postaci struktury Mat. Liczba struktur znajdujących się w module jest naprawdę duża.

Moduł ten zawiera także metody algebry liniowej i statystyki. Pozwalają one dokonywać przekształceń algebraicznych na strukturach wejściowych. Przykładem takich metod mogą być operacje na macierzach. Zaimplementowana została możliwość ich dodawania, odejmowania, dzielenia, mnożenia, wyliczania normy, wyliczania macierzy odwrotnej i jeszcze kilka innych operacji.

Bardzo ważną funkcją modułu jest możliwość zapisywania oraz odczytywania wszystkich zdefiniowanych w nim struktur danych w postaci pliku XML lub YAML. Daje to ogromnie możliwości w przechowywaniu utworzonych struktur. Ponadto w module tym zawarte są funkcje służące obsłudze błędów.

3.1.2 Imgproc

Moduł zawierający funkcje z dziedziny filtrowania obrazów oraz ich przekształceń. Służą one do wykonywania przeróżnych liniowych i nielinowych operacji na dwuwymiarowych obrazach. Obrazy przetrzymywane są w postaci struktury danych Mat.

Filtry operują najczęściej na każdym pikselu obrazu wejściowego i produkują odpowiedni wynik na obrazie wyjściowym. W takim przypadku obraz wyjściowy jest tego samego rozmiaru co obraz początkowy. Kiedy obraz wejściowy składa się z kilku kanałów, każdy z nich przekształcany jest niezależnie. Nowy obraz również będzie składał się z takiej samej liczby kanałów co obraz wejściowy.

Poza wymienionymi wyżej, moduł zawiera także funkcje do wykonywania transformacji geometrycznych obrazów dwuwymiarowych. Nie wyliczają one nowych wartości pikseli tylko kopiują wartość jednego piksela w drugie miejsce. Istnieją jeszcze funkcje do analizy kształtów, ich opisywania oraz do wyliczania histogramów obrazów. Więcej informacji o module, a także przykłady zastosowań można znaleźć w [3].

3.1.3 Highgui

Funkcje zawarte w module umożliwiają tworzenie interfejsów użytkownika. Funkcjonalność podstawowa biblioteki OpenCV może być używana bez jakiegokolwiek interfejsu, jednak czasem potrzebne jest szybkie pokazanie rezultatów. Nie trzeba w tym celu dodawać ani wykorzystywać innych bibliotek, wystarczy funkcjonalność modułu Highgui.

Przeznaczeniem modułu jest tworzenie prostych okien i wyświetlanie w nich obrazów lub klipów wideo. Użytkownik nie musi martwić się o wywoływanie funkcji przeładowania zawartości okna. Do okien można dodawać

przyciski i suwaki służące zmianie parametrów programu lub algorytmu. Poza tym istnieje możliwość definiowania operacji dla myszy i klawiatury.

Poza tworzeniem okien i wypełnianiem ich treścią, moduł daje możliwość dokonywania zapisu i odczytu obrazów oraz sekwencji wideo z dysk twardego.

3.1.4 Video

Moduł odpowiedzialny jest za przetwarzanie sekwencji wideo. Funkcje w nim zawarte służą do analizowania ruchu i śledzenia obiektów na filmach. Przykładem zastosowania może być odnajdywanie ruchomych elementów w filmie i ich klasyfikowanie.

Dostępne funkcje to przede wszystkim filtr Kalmana oraz obliczanie i usuwanie tła z obrazu. Tło to nieruchoma część obrazu, która powinna być usunięta przy analizie ruchu na obrazie.

3.1.5 Calib3d

Zawartość modułu to przede wszystkim funkcje do kalibracji kamery z trójwymiarową rzeczywistością, projekcji punków trójwymiarowych na powierzchni dwuwymiarowej oraz wyliczaniu nowych punktów po odpowiednich przekształceniach. Funkcje te przeznaczone są do zadań z dziedziny stereowizji i widzenia komputerowego.

Dzięki modułowi możliwe jest odtwarzanie przestrzeni trójwymiarowej na podstawie obrazu z dwóch kamer. Odpowiednie funkcje służą ustaleniu pozycji kamer względem siebie, ich parametrów oraz obliczaniu dysparycji obrazów wejściowych. Obliczenia te mają na celu ustawienie osi optycznych obu kamer równolegle względem siebie. Tak przetworzony obraz może zostać przeniesiony do wirtualnej rzeczywistości.

3.1.6 Features2d

Ten moduł zawiera funkcje służące wykrywaniu cech charakterystycznych obrazu. Mogą one posłużyć do odszukiwania obrazu referencyjnego na innym obrazie. Obraz wejściowy wcale nie musi być identyczny z fragmentem obrazu przeszukiwanego. Może on być poddany pewnym przekształceniom, a funkcje zawarte w bibliotece umożliwiają jego odnalezienie.

Każdy obraz można opisać za pomocą odpowiednich deskryptorów. Deskryptory te służą do jednoznacznego opisania obrazu tak, by był on rozpoznawany pośród innych obrazów. Dostępne w module deskryptory to m. in. SURF, FAST, ORB i STAR.

Dodatkowo, moduł umożliwia konwersję punktów kluczowych odnalezionych różnymi deskryptorami. Daje to możliwość porównywania działania różnych algorytmów detekcji punktów kluczowych. Moduł umożliwia też nanoszenie punktów kluczowych na obraz, co przydaje się przy przeglądaniu wyników.

3.1.7 Objdetect

Moduł poświęcony jest detekcji obiektów na obrazach. Jego funkcje to przede wszystkim nauka klasyfikatora opartego o klasyfikator Haar. Klasyfikator ten, w zależności od złożoności obiektu, do nauki wymaga podania kilkuset do kilku tysięcy przykładów.

Dzięki nauce klasyfikatora, można rozpoznawać przeróżne obiekty na obrazach. Obiekty te mogą być widoczne z różnej perspektywy i przy różnym oświetleniu. Nie muszą być one identyczne, np. jeśli nauczymy klasyfikator jak wygląda kubek z uchem to rozpoznawać on będzie nie tylko ten konkretny kubek, ale też inne podobne.

3.1.8 Ml

Bardzo obszerny moduł, poświęcony uczeniu maszynowemu. Zawarte w nim są funkcje i klasy statystycznych klasyfikatorów oraz funkcje poświęcone grupowaniu danych. Wszystkie klasyfikatory dziedziczą po jednej klasie, żeby ujednolicić i uprościć ich użytkowanie.

Dostępne klasyfikatory to m. in.: klasyfikator Bayesa, K-najbliższych sąsiadów, Drzew decyzyjnych, drzew przycinanych i drzew losowych. Do każdego z nich istnieją funkcje uczące klasyfikator, ustawiające odpowiednie dla nich parametry oraz funkcje testujące.

Poza statystycznymi klasyfikatorami w module zawarte są mechanizmy do nauki sieci neuronowych. Umożliwiają one tworzenie, ćwiczenie oraz testowanie sztucznych sieci neuronowych. Przy ich zastosowaniu niektóre zadania klasyfikacji mogą być wykonywane szybciej i bardziej precyzyjnie.

3.1.9 Flann

Moduł FLANN (Fast Library for Approximate Nearest Neighbors) to zbiór funkcji i algorytmów zoptymalizowanych do szybkiego wyszukiwania w dużych zestawach danych. Więcej informacji o tym, czym jest FLANN można znaleźć tutaj [12].

Wewnętrzne funkcje modułu przeznaczone są do wczytywania zestawu danych, ich przeszukiwania oraz pobierania wyników.

3.1.10 Gpu

W module tym znajdują się funkcje i klasy wykorzystujące procesory graficzne do obliczeń. Moduł wykorzystuje tylko i wyłącznie technologię CUDA, więc obsługuje tylko karty graficzne NVidii. Funkcje modułu to najczęściej przerobione wersje funkcji z innych modułów w taki sposób, aby wykorzystywały moc kart graficznych.

Moduł został zaprojektowany tak, żeby łatwo było go wykorzystywać i nie wymaga znajomości CUDY. Warto jednak zapoznać się z zasadami obliczeń na kartach graficznych, by zrozumieć jakie dają możliwości, ale przede wszystkim jakie ograniczenia są na nie nałożone.

Kompilacja modułu odbywa się tylko na żądanie użytkownika. Żeby korzystać z modułu, trzeba przed kompilacją zaznaczyć odpowiednią flagę.

3.2 Wielowątkowe przetwarzanie obrazów w bibliotece OpenCV

Już od wczesnych wersji biblioteki, autorzy zauważyli, że operacje na obrazie dają się często dobrze zrównoleglić. Ciężko przewidzieć jednak do jakich celów będzie wykorzystywana biblioteka. Dla przykładu, jeśli użytkownik przetwarzałby obraz biblioteki o rozdzielczości kilku milionów piekseli, to zysk ze zrównoleglenia byłby widoczny gołym okiem. Jeśli natomiast obraz miałby zaledwie kilkaset pikseli, to nakład narzucony przy komunikacji i podziale pracy mógłby przerastać zysk. Dlatego do kwestii zrównoleglania funkcji podchodzono ostrożnie. Tylko najbardziej złożone obliczeniowo funkcje zostały napisane w sposób umożliwiający wykorzystanie dobrodziejstw procesorów wielordzeniowych. Autorzy pozostawili więc zrównoleglanie obliczeń programistom. Na dzień dzisiejszy, spośród wszystkich pięciuset funkcji tylko kilka ma zaimplementowaną obsługę wielu wątków.

Na początku do zrównoleglania obliczeń wykorzystywano otwartą bibliotekę OpenMP. Od wersji OpenCV oznaczonej numerem 2.2 wykorzystywane jest już tylko Intel TBB. Niestety, żadna z oficjalnie udostępnionych i skompilowanych wersji biblioteki nie korzysta z wielowątkowości. Żeby z niej korzystać należy ściągnąć ze strony źródła biblioteki OpenCV i skompilować je w odpowiedni sposób. Proces ten został opisany dokładniej w dziale 4.2. Dopiero wtedy programy, w których użyto natywnie zrównoleglonych funkcji będą działać szybciej. Wszystko to dlatego, że twórcy biblioteki nie chcą narzucać ani sposobów ani bibliotek zrównoleglania kodu.

3.2.1 Spis funkcji zrównoleglonych

Lista funkcji, które zostały zrównoleglone przez autorów biblioteki nie jest nigdzie opublikowana. Praca inżynierska wymagała więc przejrzenia biblioteki OpenCV w poszukiwaniu tych funkcji. W tym celu pobrane zostały źródła biblioteki. Poszukiwane funkcje wyróżniały się tym, że znajdowały się w nich dyrektywy dla kompilatora o następującej treści :

```
#ifdef HAVE_TBB
```

Przykładem odnalezionego pliku jest „surf.cpp" będący składową modułu features2d. Interesujący nas fragment pliku pokazany jest na rys 3.1

```
920     if ( N > 0 )
921     {
922 #ifdef HAVE_TBB
923         cv::parallel_for(cv::BlockedRange(0, N),
924                          cv::SURFInvoker(&params, keypoints, descriptors, img, sum) );
925 #else
926         cv::SURFInvoker invoker(&params, keypoints, descriptors, img, sum);
927         invoker(cv::BlockedRange(0, N));
928 #endif
929     }
930
931
932     /* remove keypoints that were marked for deletion */
933     for ( i = 0; i < N; i++ )
934     {
935         CvSURFPoint* kp = (CvSURFPoint*)cvGetSeqElem( keypoints, i );
936         if ( kp->size == -1 )
937         {
938             cvSeqRemove( keypoints, i );
939             if ( _descriptors )
```

Rysunek 3.1: Odnaleziona dyrektywa kompilatora wskazująca na natywne zrównoleglenie funkcji

Po przejrzeniu wszystkich plików powstała następująca lista funkcji:

- ann_mlp::CvANN_MLP() (wykorzystywane w funkcji wewnętrznej operator())
- SurfFeatureDetector::detect() - funkcja służąca odnajdywaniu punktów kluczowych na obrazie;
- SurfDescriptorExtractor::extract() - funkcja służąca tworzeniu detektora pliku graficznego; detektor umożliwia odnajdywanie podobieństw w obrazach oraz szukanie obrazu w obrazie po przekształceniach;
- objdetect::cascadedetect() - służy do wykrywania obiektów na obrazie, opisanych wcześniej klasyfikatorem haar;

- objdetect::haar.haartraining() - przygotowuje klasyfikator haar na podstawie wielu obrazów tego samego obiektu;
- objdetect::llatentSVM.detector() - funkcja tworząca klasyfikator SVM, opisujący obiekty na zdjęciach;
- calib3d::solvepnp() - oblicza pozycję obiektu 3d na zdjęciu 2d zadanego w postaci punktów;

Jak widać funkcji tych nie jest zbyt wiele. Odnalezione funkcje dotyczą zazwyczaj operacji wymagających naprawdę bardzo dużej liczby obliczeń. Liczba odnalezionych funkcji jest jednak wystarczająca do realizacji projektu. Napisanie programów wykorzystujących każdą z funkcji wymagało zaznajomienia się z ich budową i funkcjonowaniem.

Rozdział 4

Realizacja projektu

Wykonanie projektu wymagało zaznajomienia się z opisywanymi wcześniej bibliotekami, tj. OpenCV oraz IntelTBB. Część implementacyjna projektu wymagała napisania dwóch programów. Pierwszy z nich miał wyliczyć jaki zysk można otrzymać, kiedy stosujemy bibliotekę OpenCV wykorzystującą TBB w stosunku do biblioteki, która nie wykorzystuje TBB. Drugi program miał przybliżyć trochę działanie samej biblioteki TBB. Zakładał więc własnoręczne zrównoleglenie wybranych funkcji biblioteki OpenCV, a także napisanie algorytmów wielowątkowych, nie wykorzystujących biblioteki OpenCV.

4.1 Użyte oprogramowanie

Do realizacji projektu użyte zostało następujące oprogramowanie:

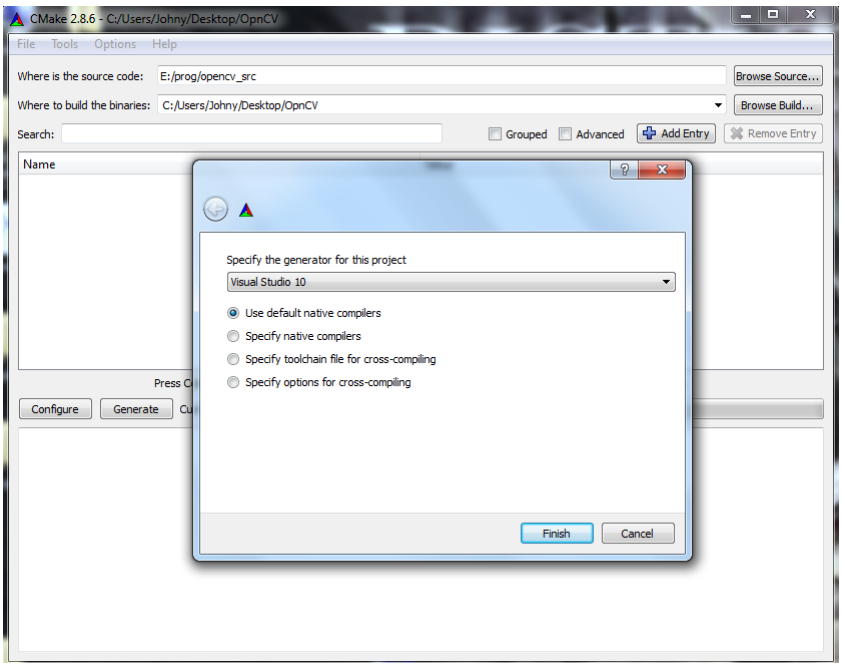
- MS Visual Studio 2010
- OpenCV 2.3.1
- Intel TBB 4.0 Update 1
- CMake 2.8.6

4.2 Kompilacja OpenCV z TBB

Dostępne na stronie projektu wersje OpenCV nie były skompilowane z flagą umożliwiającą wykorzystanie biblioteki TBB. Dlatego właśnie należało pobrać źródła OpenCV i przystąpić do odpowiedniego ich skompilowania. Poniżej znajdują się kroki opisujące wspomnianą kompilację.

1. Najpierw pobrany został kod źródłowy biblioteki OpenCV ze strony oraz skompilowana bibliotek Intel TBB.

2. Teraz w programie CMake wskazana została ścieżka do biblioteki i naciśnięty przycisk „configure” (4.1).



Rysunek 4.1: Kompilacja biblioteki OpenCV krok 1

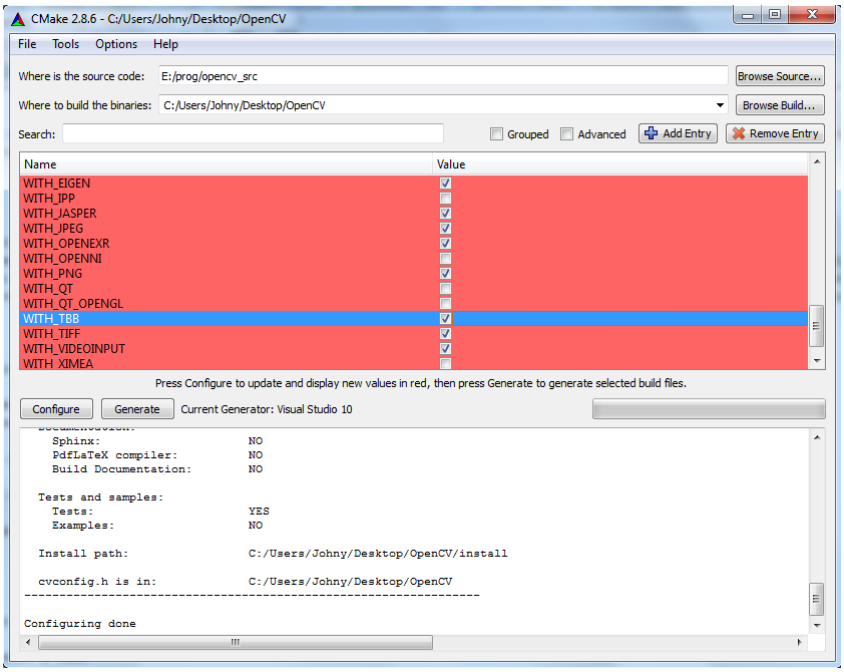
3. Wybrano opcję Visual Studio 10 i naciśnięto przycisk „Finish”

4. Kiedy pokazała się lista dostępnych opcji kompilacji, zaznaczono na niej „WITH_TBB” (4.2).

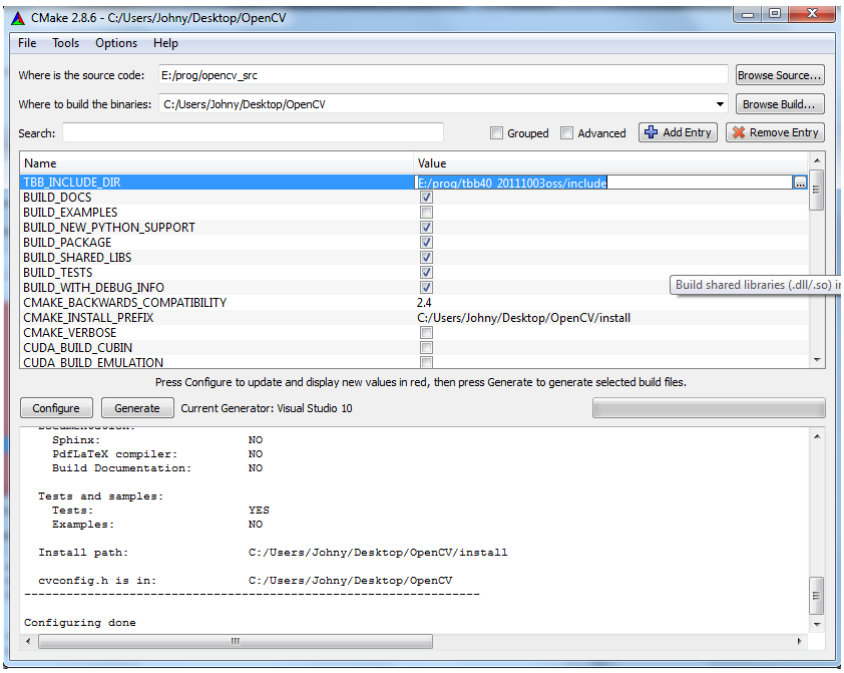
5. Po kolejnym wciśnięciu przycisku „configure”, podana została ścieżka do biblioteki Intel TBB (4.3).

6. Po raz trzeci wciśnięty został przycisk „configure”, tym razem plik projektu dla Visual Studio został poprawnie utworzony.

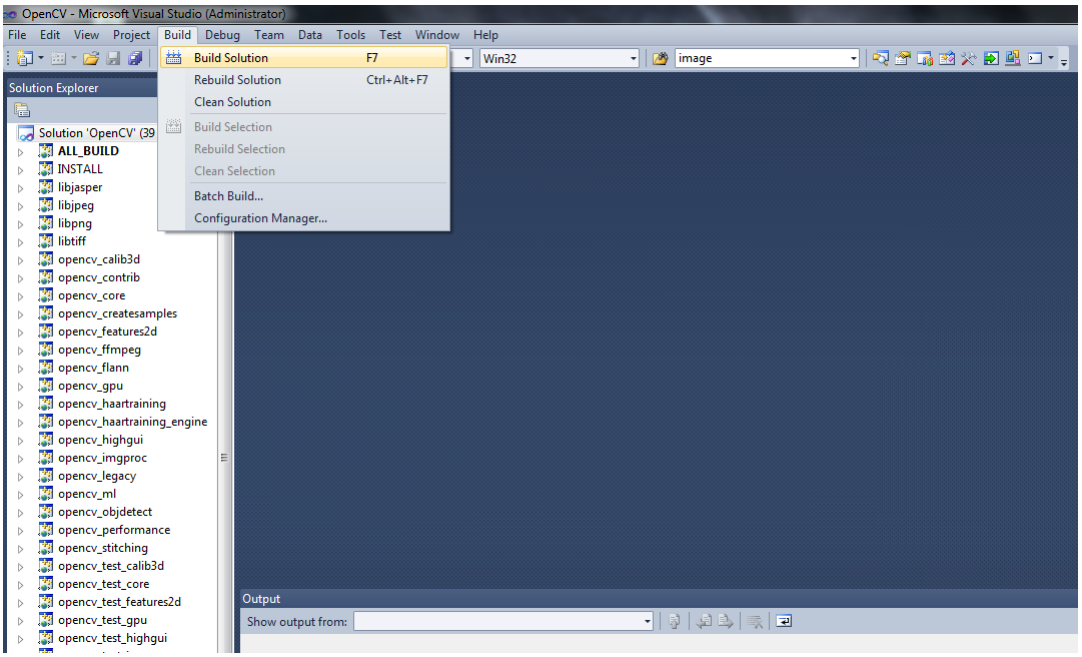
7. Uruchomiono powstały projekt i dokonano kompilacji wybierając opcję „Build Solution” z menu „Build” (4.4).



Rysunek 4.2: Kompilacja biblioteki OpenCV krok 2



Rysunek 4.3: Kompilacja biblioteki OpenCV krok 3



Rysunek 4.4: Kompilacja biblioteki OpenCV krok 4

4.3 Specyfikacja funkcjonalna programu pierwszego

4.3.1 Główny algorytm programu

Program pierwszy miał za zadanie przetestować jak wydajne jest zrównoleglenie wbudowane w bibliotekę OpenCV. Testowanie odbywa się na dwóch wersjach programu. Jedna z nich wykorzystuje podstawową wersję biblioteki OpenCV, druga zaś wersję OpenCV z TBB. Dzięki możliwości ustawiania w MS Visual Studio różnych konfiguracji dla kompilacji, możliwy jest prosty wybór kompilowanej wersji programu. Wystarczy ustawić zakładkę „Solution Configuration" na TBB lub NO_TBB a program zostanie skompilowany w odpowiedniej wersji. Konfiguracje te różnią się ustawieniami ścieżek do biblioteki OpenCV. Jedna z nich wskazuje na zwykłą bibliotekę, a druga na bibliotekę wykorzystującą Intel TBB.

Program nie wymaga żadnych danych wejściowych, po uruchomieniu od razu przystępuje do wykonywania testów. Na samym początku wyświetla informację, czy wykorzystuje TBB, czy nie. Po każdym zakończonym teście wyświetla jego nazwę oraz czas jaki był potrzebny do ukończenia testu. Gdy

program skończy wykonywać wszystkie testy, zadaje użytkownikowi dwa pytania. Najpierw pyta się, czy ma wyświetlić obrazy, które wygenerował. Jeśli użytkownik odpowie twierdząco na pytanie, jego oczom ukażą się odpowiednio zatytułowane okna, w których można będzie obejrzeć, to co zostało wykonane na plikach wejściowych. Po zakończeniu oglądania obrazów lub negatywnej odpowiedzi na pytanie pierwsze, program zadaje użytkownikowi pytanie o ponowne uruchomienie testów. Jeśli odpowiedź będzie twierdząca wykona wszystkie testy ponownie, w przeciwnym wypadku zakończy swoje działanie.

4.3.2 Funckcje OpenCV wybrane do testów

Do testowania wybrane zostały w miarę różne funkcje tak, by można było zobaczyć w jakim stopniu udało się je zrównoleglić programistom biblioteki. Oto lista funkcji, które są testowane w programie :

- SurfFeaturesDescriptor.detect()
- CvGBTrees.train()
- CascadeClassifier.detectMultiScale()
- CvANN_MLP.train()
- konstruktor StereoBM()

Z wymienionych powyżej funkcji dwie nie operują na obrazach, przy czym do jednej z nich można stworzyć interpretację graficzną. Funkcje te to CvGBTrees.train() oraz CvANN_MLP.train(). Obie operują na danych nie będących bezpośrednio obrazami. Do funkcji CvANN_MLP.train() napisana została wizualizacja, by zobaczyć w jaki sposób dane zostały zaklasyfikowane.

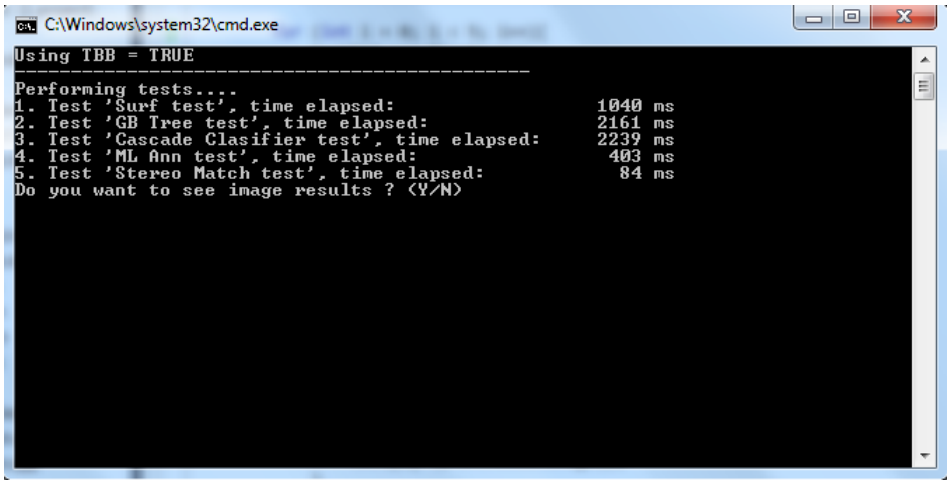
Pozostałe trzy funkcje na podstawie obrazów wejściowych produkują obraz wyjściowy. Obrazy wyjściowe można oglądać w programie po ukończeniu testowania wszystkich funkcji.

4.3.3 Dane wyjściowe

Program na bieżąco wypisuje na ekran rezultaty wykonywanych testów, a po ich zakończeniu prosi użytkownika o interakcję. Wynik testu opisany jest za pomocą numeru testu, jego nazwy oraz czasu, jaki potrzebny był na jego przeprowadzenie. Linia danych wyjściowych tworzona jest według następującego schematu:

```
<nr> Test <nazwa>, time elapsed: <czas_wykonywania>
```

Rysunek 4.5 pokazuje jak wygląda okno uruchomionego programu i w jaki sposób prezentowane są wyniki testów.

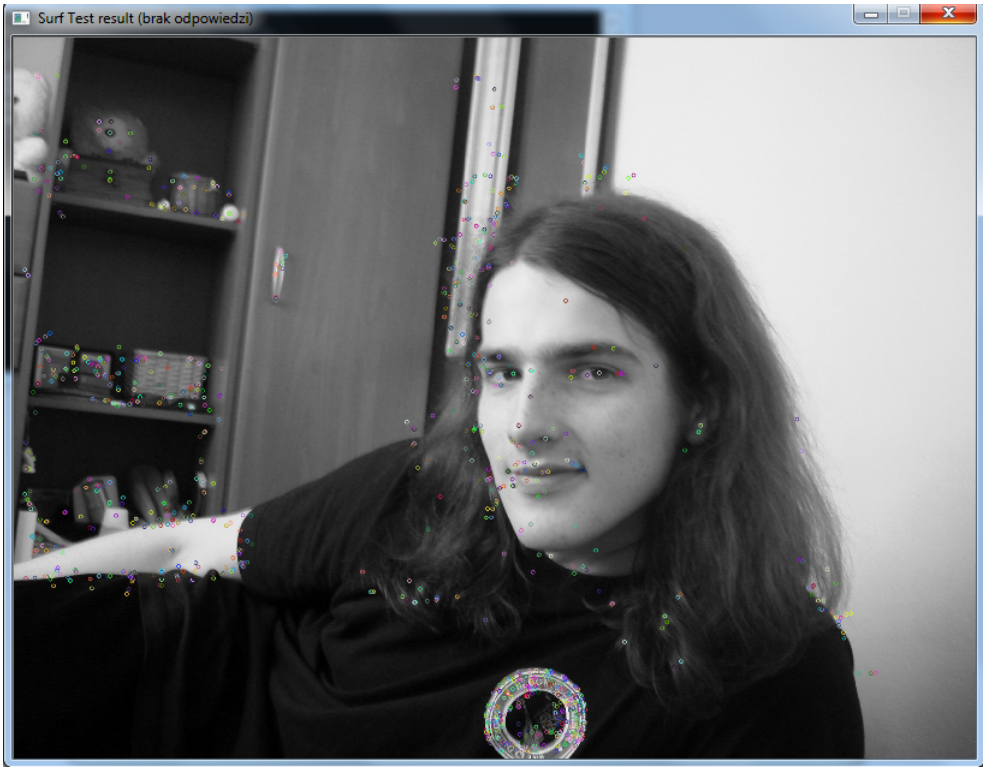


Rysunek 4.5: Okno ukazujące wyniki prezentowane przez program pierwszy

Poza czystymi danymi statystycznymi istnieje możliwość podglądu obrazu wyjściowego. Jest to jednak funkcjonalność opcjonalna, służąca raczej poglądowej ocenie, czy algorytmy działają poprawnie i czy dają identyczne rezultaty. Jeśli użytkownik zechce obejrzeć wyniki graficzne, to wystarczy, że po zakończeniu testów odpowie twierdząco na pytanie o ich wyświetlenie. Zostanie wtedy pokazane okno z wynikami. Przykładowe okno widoczne jest na rysunku 4.8

4.3.4 Obsługa błędów

Program oferuje prostą obsługę błędów. Przy wykonywaniu testów sprawdzone zostaje czy istnieją pliki wejściowe. Jeśli jakiś plik nie będzie dostępny dla programu, to program wyświetli odpowiedni komunikat o błędzie, a test zostanie pominięty. To samo dotyczy się obrazów wyjściowych. Jeśli którykolwiek obraz nie będzie chciał się otworzyć lub w czasie jego tworzenia powstanie błąd, to program wygeneruje odpowiedni komunikat. W momencie wyświetlania wszystkich obrazów wynikowych, obraz którego nie udało się utworzyć zostanie najzwyczajniej pominięty.



Rysunek 4.6: Okno pokazujące graficzny efekt pracy programu; Widoczny rysunek pokazuje działanie funkcji SurfFeaturesDescriptor.detect() po nałożeniu punktów kluczowych na obraz bazowy

4.4 Specyfikacja implementacyjna programu pierwszego

4.4.1 Główny algorytm programu

Główny algorytm programu jest dosyć prosty w swojej budowie. Program nie przetwarza żadnych argumentów z wejścia. Najpierw tworzone są obiekty testów. Następnie obiekty te są rzutowane na klasę Test i układane w wektorze. Zaraz potem program wyświetla informację, czy używa biblioteki Intel TBB, czy też nie. Po wyświetleniu tej informacji przechodzi do pętli głównej.

Pętla główna to pętla typu „do...while". Gwarantuje ona, że kod w niej znajdujący wykona się przynajmniej raz. Wewnątrz znajdują się dwie pętle „for". Pierwsza z nich iteruje po elementach wektora testów i wykonuje na każdym elemencie operację performTest(). Operacja ta zależna jest od obiektu, na którym jest wykonywana. Każdy z obiektów podczas wykonywania tej operacji dokonuje pomiaru czasu. Czas ten jest przetrzymywany w zmiennej prywatnej dla każdego obiektu osobno. Po zakończeniu testu wyświetlany jest wynik. Program główny pobiera go od testowanego obiektu przy pomocy funkcji getResult(). Wyniki wyświetlane są w taki sposób, żeby czasy znajdywały się zaraz pod sobą w jednej kolumnie.

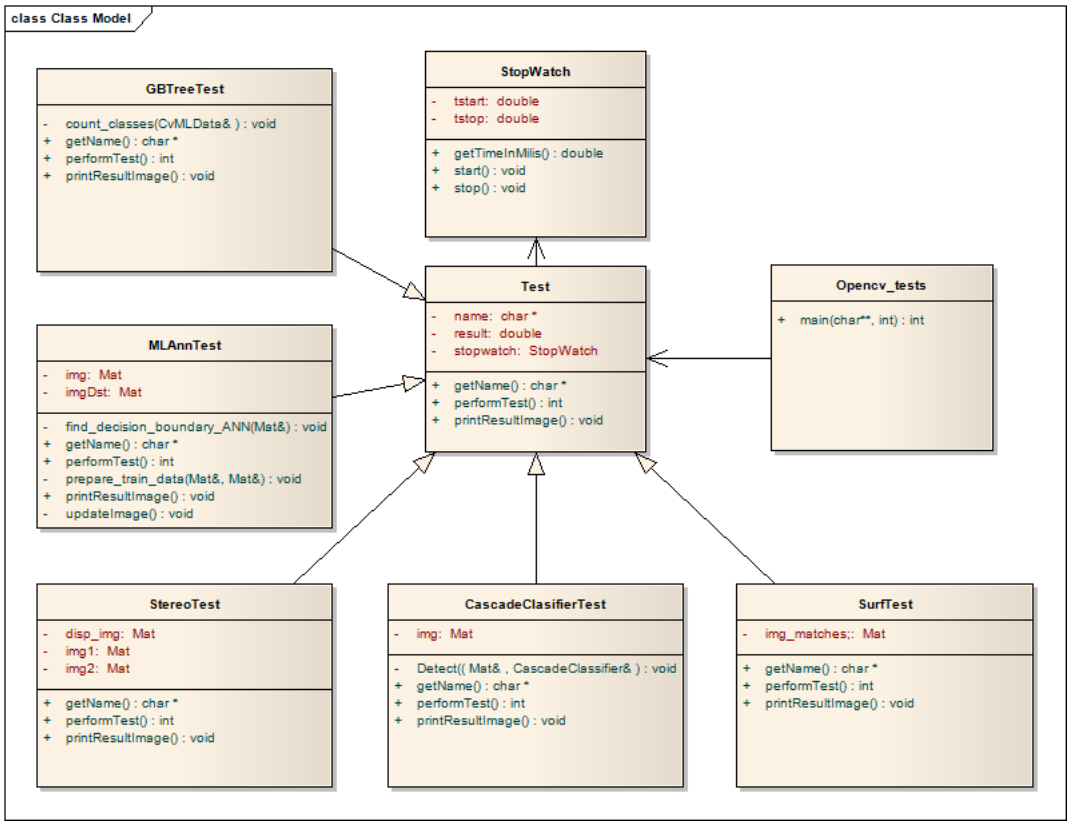
Po przejściu przez wszystkie elementy wektora, program wyświetla zapytanie i oczekuje na wprowadzenie znaku z klawiatury. Jeśli znak ten jest literą „Y", to przystępuje do wyświetlenia obrazów utworzonych przez poszczególne algorytmy. W tym celu dokonuje ponownej iteracji po wszystkich elementach wektora testów i wykonuje dla każdego z nich operację printResultImage(). Jeśli natomiast wprowadzony znak będzie dowolnym innym znakiem, program pomija wyświetlanie obrazów.

Po zakończeniu wyświetlania obrazów bądź też po pominięciu tego kroku, program ponownie wymaga interwencji użytkownika. Zostaje wyświetlone zapytanie, czy powtórzyć wszystkie testy od nowa. Jeśli użytkownik wprowadzi literę „Y", to następuje powrót do początku pętli „do...while". Pętla ta jest powtarzana tak długo aż odpowiedź na ostatnie pytanie nie będzie znakiem 'Y'. Po wyjściu z pętli program kończy swoje działanie.

4.4.2 Podział na moduły

Program zbudowany jest w sposób modularny, co znacznie ułatwia jego rozbudowę i edycję. Dołożenie kolejnych testów sprowadza się do utworzenia odpowiedniej klasy dziedziczącej po klasie Test. Nowa klasa musi implementować trzy metody: performTest(), getResult() oraz getImageResult(). Wystarczy potem w głównym programie zadeklarować nową klasę, wywołać jej

konstruktor i dodać ją do wektora testów. Podział na moduły widoczny jest na rysunku 4.7.



Rysunek 4.7: Podział na moduły pierwszego programu

4.4.3 Opis modułów

Program składa się z ośmiu modułów. Pięć z nich to moduły testów, pozostałe trzy to moduł główny, moduł do pomiaru czasu oraz interfejs klasy test. Krótki opis każdego z nich znajduje się poniżej.

- Opencv_tests - główny moduł programu, jego opis doskonale oddaje rozdział 4.4.1
- Test - klasa macierzysta dla testów, jest tak jakby interfejsem, za pomocą którego program główny komunikuje się z pozostałymi testami; zawarte w niej są wszystkie operacje, które potrzebne są do przeprowadzenie testu i zebrania jego wyniku;

- StopWatch - klasa stoper służąca do pomiaru czasu jaki upłynął pomiędzy dwoma fragmentami kodu; klasa ta wykorzystuje pomiar czasu z biblioteki OpenCV, na chwilę obecną zwraca czas z dokładnością 1 ms
- GBTreeTest - klasa testująca funkcję CvGBTrees.train(), przygotowuje odpowiednie dane wejściowe i przeprowadza trening drzew;
- MLAnnTest - klasa testująca funkcję CvANN_MLP.train(), uczy sieć neuronową, a następnie ją odpytuje;
- StereoTest - klasa testująca konstruktor StereoBM(), wczytuje ona dwa obrazy, a następnie oblicza dla nich dysparycję;
- CascadeClasifierTest - klasa testująca funkcję CascadeClassifier.detectMultiScale(); wczytuje ona obraz i odszukuje na nim twarze;
- SurfTest - klasa testująca funkcję SurfFeaturesDescriptor.detect(); odszukuje punkty kluczowych na obrazie referencyjnym;

Ostatnie pięć klas dziedziczy po klasie Test. Każda z tych klas dokonuje pomiaru czasu określonych funkcji wewnętrznych i zapamiętuje rezultat w zmiennej result. Metody klasy macierzystej „Test" są przesłaniane przez prywatne odpowiedniki klas dziedziczących.

4.5 Specyfikacja funkcjonalna programu drugiego

4.5.1 Opis programu

Drugi program, który powstał w ramach projektu inżynierskiego, miał na celu testowanie algorytmów własnych zrównoleglonych przy pomocy biblioteki Intel TBB. Podczas pisania podjęta została próba zrównoleglenia funkcji SurfFeaturesDescriptor.detect(). Ponadto zrównoleglony został niezależny algorytm dokonujący erozji obrazu oraz algorytm wyliczający momenty.

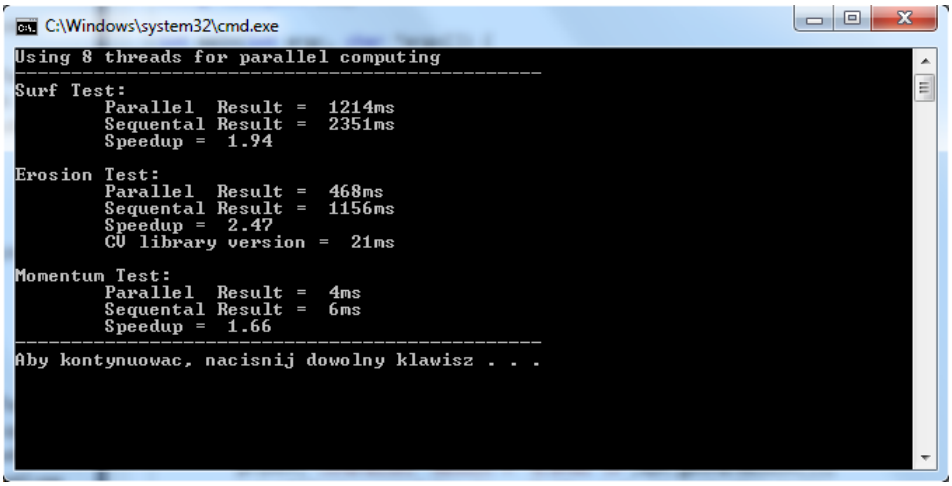
Tak samo jak program pierwszy, drugi nie przyjmuje żadnych parametrów. Po uruchomieniu od razu przechodzi do testów, a wyniki są od razu wyświetlane na ekranie. Każdy test składa się tak naprawdę z dwóch testów. Pierwszy z nich jest sekwencyjny, drugi natomiast zrównoleglony. Tym razem prezentacja wyników jest nieco inna. Poza czasem potrzebnym na wykonanie algorytmów podawany jest zysk ze zrównoleglenia. Po zakończeniu testów można obejrzeć wygenerowane przez program obrazy.

Kiedy pierwsze trzy testy zostaną ukończone, użytkownik może obejrzeć specjalny test video. Polega on na pokazaniu różnic w sekwencyjnym i równoległym nakładaniu filtru na dwa obrazy pochodzące z kamery video.

Podczas pisania programu drugiego wykorzystane zostało źródło aplikacji napisanej przez dr inż. Marcina Iwanowskiego. Wykorzystane z niej zostały dwa algorytmy: do wykonywania erozji obrazu oraz do obliczania momentów.

4.5.2 Dane wyjściowe

Dane prezentowane przy pracy programu są podobne do danych prezentowanych przez pierwszą aplikację. Tym razem nie została tutaj uwzględniona liczba porządkowa dla testów. Na ekranie prezentowana jest po prostu nazwa testu, wynik przebiegu sekwencyjnego, wynik przebiegu równoległego oraz przyspieszenie. Rysunek 4.8 pokazuje jak wygląda okno z wynikami.



Rysunek 4.8: Okno ukazujące wyniki prezentowane przez program drugi

4.5.3 Obsługa błędów

Zupełnie jak w przypadku programu pierwszego, tutaj także występuje prosta obsługa błędów. W przypadku braku dostępu do obrazów referencyjnych wyświetlony zostaje odpowiedni komunikat, a test zupełnie pominięty. Dodatkową funkcjonalnością jest porównywanie obrazów wyjściowych. Jeśli obraz utworzony w algorytmie sekwencyjnym różni się od obrazu utworzonego w algorytmie zrównoleglonym program zgłasza to obok wyniku testu i proponuje obejrzenie obu obrazów wyjściowych. Takie sprawdzanie ma na

celu ukazanie, że algorytmy wykonują dokładnie taką samą pracę w różnym czasie.

4.6 Specyfikacja implementacyjna programu drugiego

4.6.1 Główny algorytm programu

Po rozpoczęciu działania, program tworzy obiekty testów. W przeciwieństwie do programu pierwszego testy nie zostają jednak upakowane do wektora. Przyczyną takiego stanu rzeczy są różnice w obiektach testowych. Nie każdy obiekt testowy zwraca dwa wyniki. Po utworzeniu obiektów program rozpoczyna wykonywanie testów.

Najpierw dokonwywany jest test własnoręcznie zrównoleglonego algorytmu SurfFeaturesDescriptor.detect(). Detekcja punktów kluczowych odbywa się najpierw dla całego obrazu. Następnie obraz zostaje podzielony na mniejsze obrazy. Liczba mniejszych obrazów jest zależna od ustalonej liczby wątków. Każdy z mniejszych obrazów jest przetwarzany równolegle. Z powodu specyficznego działania funkcji „detect()", mniejsze obrazy muszą się zazębiać. Liczba pikseli nachodzących na siebie została dobrana eksperymentalnie. Dopiero wtedy rezultaty obu algorytmów są identyczne. Po skończeniu testów zapisane zostają odpowiednie pomiary czasów.

Drugi wykonywany jest test erozji obrazu. Najpierw dokonywana jest erozja algorytmem sekwencyjnym, a później erozja zrównoleglona. W tym przypadku obraz podzielony jest na linijki. Każda linijka może być przetwarzana niezależnie, gdyż pozwala na to stworzony algorytm. Obraz wejściowy nie ulega zmianie, dlatego pojedyncze piksele obrazu wyjściowego mogą być na bieżąco zapisywane. Następnie, dla porównania wyników, dokonywana jest erozja funkcją dostępną w bibliotece OpenCV

Kolejnym testem wykonywanym w programie jest liczenie momentów obrazu. Tutaj także obraz dzielony jest na linijki. O ile przetwarzanie obrazu wejściowego może być wykonywane równolegle bez żadnych komplikacji o tyle zapisywanie wyników już nie. Zapis do tablicy momentów musi być kontrolowany tak, by dwa wątki nie próbowały nadpisywać tego samego pola w jednej chwili. W tym celu wykorzystany został mechanizm zamka prostego.

Po zakończeniu testów program wyświetla pytanie o pokazanie otrzymanych wyników. Jeśli użytkownik wciśnie przycisk 'Y' na ekranie ukazują się obrazy wynikowe wszystkich wykonanych testów. Następnie, kiedy wciśnięty zostanie dowolny klawisz, to zadawane jest pytanie o test Video. Test ten jest testem ciągłym, dlatego przed jego rozpoczęcie wymagana jest interwencja

użytkownika. Jeśli użytkownik wciśnie przycisk 'Y' to program rozpoczyna wyświetlanie obrazu z kamery w dwóch oknach w sposób sekwencyjny. W oknie programu na bieżąco wyświetlana jest ilość klatek na sekundę. Test trwa tak długo, aż użytkownik wciśnie przycisk 'ESC' na klawiaturze. Wtedy rozpoczyna się wyświetlanie tych samych obrazów, ale z wykorzystaniem biblioteki Intel TBB. Kiedy znowu przyciśnięty zostanie przycisk ESC to zarówno ostatni test jak i cały program kończą swoje działanie.

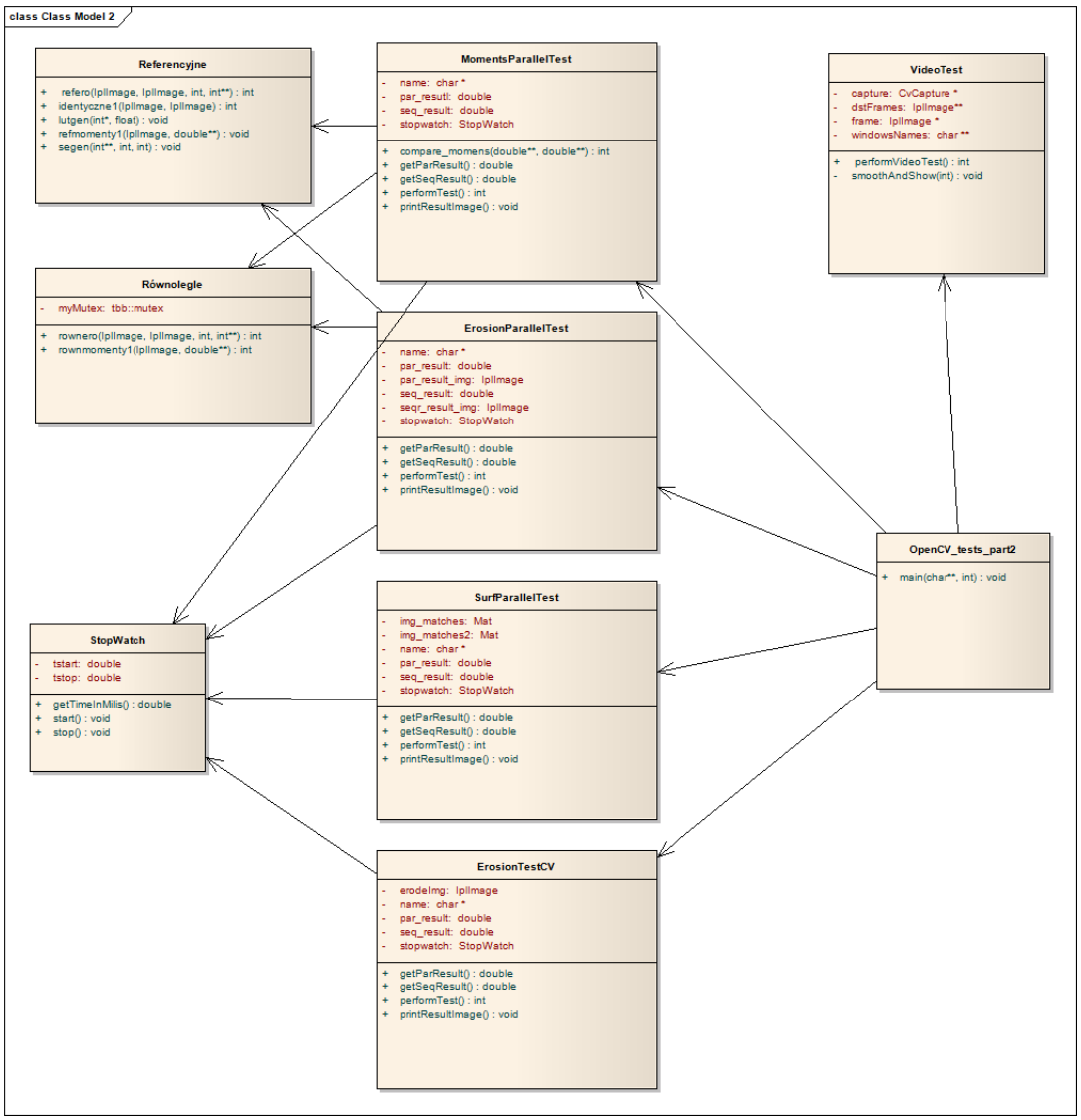
4.6.2 Podział na moduły

W przypadku programu drugiego również występuje budowa modularna. Tym razem operacje testów wykonywane są bezpośrednio przez program główny. Dlatego właśnie dokładanie nowego testu wymaga większej ingerencji w kod niż w pierwszym programie. Poza stworzeniem nowej klasy testu i utworzeniem jej instancji w programie głównym, trzeba jeszcze wywołać funkcję wyświetlającą wyniki. Należy także dodać w odpowiednim miejscu funkcję umożliwiającą podgląd obrazu wynikowego. Dokładny podział na moduły widoczny jest na rysunku 4.9.

4.6.3 Opis modułów

Program drugi zbudowny jest z dziewięciu modułów. Pięć modułów zawiera testy. Z pozostałych czterech, dwa to moduł główny i moduł pomiaru czasu. Ostatnie dwa to zbiór funkcji do wykonywania erozji oraz obliczania momentów obrazu. Opis każdego z modułów znajduje się poniżej:

- OpenCV_tests_part2 - główny moduł programu, w nim znajduje się główna pętla; opis w dziale 4.6.1
- Referencyjne - zbiór funkcji sekwencyjnych służących do obróbki obrazów napisany przez dr inż. Marcina Iwanowskiego
- Równoległe - zbiór funkcji z modułu Referencyjne przerobionych by wykorzystywały zrównoleglenie przy pomocy biblioteki Intel TBB
- StopWatch - dokładna kopia modułu z pierwszego programu o takiej samej nazwie; służy do pomiaru czasu w dwóch różnych miejscach programu;
- MomentsParallelTest - klasa testująca wyliczanie momentów dla obrazu w sposób równoległy i sekwencyjny
- ErosionParallelTest - j.w. tylko testowane jest wykonywanie erozji na obrazie



Rysunek 4.9: Podział na moduły drugiego programu

- ErosionTestCV - test sekwencyjnego wykonywania erozji na obrazie przy pomocy biblioteki OpenCV
- SurfParallelTest - klasa testująca sekwencyjną i równoległą wersję funkcji SurfFeaturesDescriptor.detect(); wersja równoległa opiera się o przetwarzanie wielu mniejszych obrazów będących częściami dużego obrazu
- VideoTest - klasa wykonująca test nakładania filtru na obraz z kamery;

4.6.4 Zrównoleglanie funkcji

Do zrównoleglania algorytmów użyta została funkcja parallel_for, będąca odpowiednikiem zwykłej pętli for. Wywołanie funkcji może nastąpić na jeden z trzech sposobów. W programie użyty był sposób zalecany przez twórców biblioteki Intel TBB jako najefektywniejszy, tzn. ten w którym przekazujemy funkcji przedział oraz specjalną klasę z metodą operator(). Opis teoretyczny znajduje się w rozdziale 2.3.1.

Jako przykład zrównoleglenia zawartego w programie opisana zostanie sekwencyjna i równoległa wersja funkcji dokonującej erozji obrazu wejściowego. Sekwencyjny algorytm wygląda następująco:

```
int refero(IplImage *imWe, IplImage *imWy, int rozm, int selem[MAXNGB][↩
    MAXNGB])
{
 uchar* wsk_we = (uchar *)imWe->imageData;
 uchar* wsk_wy = (uchar *)imWy->imageData;
 int ox, oy, oyy, oxx;
 int min;
 PETLA_OBR_SZ(imWe,przes,poz,MG)
         {   min = 255;
             PETLA_MASKA_SZ(ox,oy,oxx,oyy,imWe->widthStep,rozm,rozm)
                 if (selem[ox][oy] && wsk_we[poz+przes+oyy+oxx] < min) min↩
                     = wsk_we[poz+przes+oyy+oxx];
             wsk_wy[poz+przes] = min;
         }
 return(1);
}
```

Funkcja dokonująca erozji obrazu otrzymuje cztery parametry: obraz wejściowy, obraz wyjściowy, rozmiar oraz tablicę. Dla wygody, wskaźniki na dane obrazu wejściowego i wyjściowego zostają przypisane do dodatkowych zmiennych:

```
    uchar* wsk_we = (uchar *)imWe->imageData;
    uchar* wsk_wy = (uchar *)imWy->imageData;
```

Następnie zainicjowane zostają pozostałe zmienne, a po nich algorytm przystępuje do wykonywania obliczeń właściwych. Pętla iteruje po każdym

pikselu obrazu, pomijając marginesy i oblicza dla niego nową wartość. Praca nie skupia się na opisie algorytmu erozji, dlatego nie zostanie on tutaj opisany. Opis algorytmu erozji można znaleźć tu [13].

Równoległa wersja algorytmu wygląda tak :

```
int rownero(IplImage *imWe, IplImage *imWy, int rozm, int selem[MAXNGB][↩
    MAXNGB]){
    tbb::task_scheduler_init init(ILE_WATKOW);
    parallel_for(blocked_range<size_t>(MG, (imWe->height - MG), 10), ↩
        Rownero_par(imWe,imWy,rozm,selem) );

    return(1);
}
```

Jak widać sama funkcja dokonująca erozji została znacznie skrócona. Przyjmuje ona takie same parametry, ale znajdują się w niej zasadniczo dwie linie kodu. Pierwsza z nich:

```
    tbb::task_scheduler_init init(ILE_WATKOW);
```

uruchamia zarządce zadań TBB dla z góry określonej ilości wątków. Druga linia :

```
    parallel_for(blocked_range<size_t>(MG, (imWe->height - MG), 10), ↩
        Rownero_par(imWe,imWy,rozm,selem) );
```

wywołuje równoległe wykonywanie pętli for. Do funkcji parallel_for przekazane zostają dwa parametry. Pierwszy z nich to zasięg blokowy, który utworzony zostaje w momencie wywołania funkcji na podstawie następującego konstruktora:

```
    blocked_range& blocked_range(begin_index, end_index, size);
```

Do konstruktora tego przekazane zostają parametry początku przedziału, końca przedziału oraz ilości elementów należących do pojedynczego bloku. Algorytm erozji nie może przetwarzać pikseli będących marginesem. Ilość linii obrazu wejściowego to imWe->height, dlatego do konstruktora zasięgu przekazane zostały wartości MG (margines) oraz imWe->height - MG (wysokość obrazu odjąć margines). Wielkość bloku została dobrana eksperymentalnie, najlepsze wyniki otrzymywane były przy 10.

Drugim parametrem przekazanym do funkcji parallel_for jest specjalnie skonstruowana klasa. Opis konstrukcji klasy znajduje się dziale w 2.3.1. W napisanym programie klasa ta została zaimplementowana w następujący sposób:

```
class Rownero_par{
    IplImage *imWe;
    IplImage *imWy;
    int rozm;
    int (*selem)[MAXNGB];

 public:

     void operator() ( const blocked_range<size_t>& r ) const {
         int ox, oy, oyy, oxx;
         int l_od = (int)r.begin();
         int l_do = (int)r.end();
         uchar* wsk_we = (uchar *)imWe->imageData;
         uchar* wsk_wy = (uchar *)imWy->imageData;
         int min;
         PETLA_OBR_SZ_FRAG(imWe,przes,poz,MG,l_od,l_do){
            min = 255;
            PETLA_MASKA_SZ(ox,oy,oxx,oyy,imWe->widthStep,rozm,rozm)
            if (selem[ox][oy] && wsk_we[poz+przes+oyy+oxx] < min){
                min = wsk_we[poz+przes+oyy+oxx];
            }
            wsk_wy[poz+przes] = min;
         }
     }

     Rownero_par(IplImage *imW, IplImage *imE,int roz, int sel[MAXNGB][↩
         MAXNGB]) : imWe(imW),imWy(imE),rozm(roz),selem(sel) { }

};
```

Jak widać konstruktor otrzymuje wskaźniki na obraz wejściowy, obraz wyjściowy, rozmiar oraz tablice. Wykonanie konstruktora przypisuje otrzymane wskaźniki do odpowiednich pól klasy. Wewnątrz znajduje się metoda operator(), która otrzymuje jako parametr wartość przedziału i na podstawie otrzymanych wartości dokonuje takich samych obliczeń jak sekwencyjna wersja algorytmu dokonywania erozji.

Biblioteka TBB zajmuje się w tym przypadku podzieleniem przedziału, który jest tak naprawdę wysokością obrazu wejściowego pomniejszona o marginesy, na bloki wielkości 10. Każdy z wątków utworzonych przez bibliotekę posiada własną instancję klasy Rownero_par. Po podzieleniu przedziału na bloki, są one wysyłane do każdego wątku w celu wykonania obliczeń. Wątki dokonują obliczeń na blokach, które są tak naprawdę pasami obrazu, a wynik zapisują na obrazie wyjściowym. Nie trzeba się tutaj martwić o zakleszczanie wątków, gdyż obraz wejściowy, z którego wyliczana jest erozja, nie ulega zmianie przez cały czas trwania algorytmu. W obrazie wyjściowym, pojedyncza linia, a nawet cały pas jest nadpisywany tylko i wyłącznie przez jeden wątek.

Rozdział 5

Analiza porównawcza wydajności przetwarzania jedno- i wielowątkowego na wybranych algorytmach przetwarzania obrazów

Utworzone programy miały testować algorytmy w różnych przypadkach. Przede wszystkim testowane miały być operacja na danych wejściowych o różnej wielkości. Obrazy, na których dokonywane miały być operacje, różniły się rozdzielczością. Dane wejściowe dla algorytmów nauki maszyn różniły się ilością elementów. Ponadto, testowane miały być przypadki dla różnej liczby wątków. Niestety, po głębszej analizie biblioteki OpenCV okazało się, że nie da się regulować liczby wątków z poziomu samej biblioteki. Istnieje funkcja, która teoretycznie powinna ustawiać maksymalną liczbę wykorzystywanych wątków, jednak jej działanie dotyczyło biblioteki OpenMP. Funkcja ta nie ma żadnego wpływu na pracę biblioteki Intel TBB, która zawsze wykorzystuje maksymalną dostępną liczbę wątków.

Wybór ilości wykorzystywanych wątków dostępny jest w bibliotece Intel TBB. Dlatego program drugi daje możliwość ich regulacji. Umożliwiło to przetestowanie zysku ze zrównoleglenia przy zmieniającej się liczbie pracujących jednocześnie wątków.

Podzespoły maszyny, na której przeprowadzane były testy:

- Procesor Intel i7 QM720, 4 x 1.6 GHz, 6 MB L3
- Pamięć: 4GB DDR3, 1333 MHz
- Dysk: 5400 RPM na interfejsie SATA

Programy testowane były na systemie Microsoft Windows 7 Professional 64 bit, jednak powinny działać, bez większych modyfikacji, na innych systemach operacyjnych.

5.1 Testy biblioteki OpenCV

Testy funkcji natywnie zrównoleglonych przeprowadzone były w kolejności od tych najbardziej złożonych obliczeniowo do tych najmniej wymagających. Każdy test wykonywany był dziesięciokrotnie, a na koniec wyliczana była średnia. Z wyników testów wyliczone zostało przyspieszenie, które udało się uzyskać. Otrzymane wyniki pogrupowane zostały według testów, których dotyczyły.

Tabela5.1 prezentuje wyniki testów funkcji SurfFeaturesDescriptor.detect().

Seria	1	2	3	4
Rozdzielczość obrazu wejściowego	1632x1224	816x612	408x306	102x77
Średni czas wykonywania sekwencyjnego	2294,7	596,4	158,5	9,7
Średni czas wykonywania równoległego	1028,4	268,5	73,8	5,2
Średnie przyspieszenie	2,23	2,22	2,15	1,87

Tablica 5.1: Wyniki testów Surf

Jak widać, wraz ze spadkiem wielkości obrazu wejściowego, spada też przyspieszenie. Oznacza to, że jakość zrównoleglenia algorytmu zależna jest od złożoności przedstawionego problemu. Należy wziąć pod uwagę fakt, iż testy przeprowadzane były przy użyciu czterech wątków. Przyspieszenie rzędu 2,23 nie jest więc wynikiem rewelacyjnym. Mimo wszystko, wyraźnie widać, że algorytm działa w sposób równoległy. Otrzymane kluczowe punkty obrazów są identyczne, świadczy to o poprawności działania algorytmu równoległego.

Drugim przeprowadzonym testem był test funkcji CascadeClassifier.detectMultiScale(). Wyniki otrzymane w trakcie wykonywania testów przedstawia tabela 5.2.

Po analizie wyników można łatwo zauważyć, że wartość współczynnika przyspieszenia jest silnie zależna od rozdzielczości obrazu wejściowego. W tym przypadku wartość waha się w przedziale 2,67 - 0.94. Spadek wartości

Seria	1	2	3	4
Rozdzielczość obrazu wejściowego	1632x1224	816x612	408x306	102x77
Średni czas wykonywania sekwencyjnego	5558,7	1371,7	309,2	11
Średni czas wykonywania równoległego	2085,5	578,6	177,5	11,7
Średnie przyspieszenie	2,67	2,37	1,74	0,94

Tablica 5.2: Wyniki testów Cascade Classifier

przyspieszenia poniżej wartości 1, przy bardzo małym obrazie do przetworzenia oznacza, że zysk ze zrównoleglenia został przewyższony przez nakład potrzebny do jego podjęcia. Przy dużym obrazie widać, że jakość zrównoleglenia w tym algorytmie jest lepsza niż w przypadku testu funkcji Surf. Dotyczy to tylko tego jednego przypadku. Wraz ze spadkiem wielkości obrazu uwypukla się wpływ części sekwencyjnej na algorytm. Należy więc uważać przy stosowaniu biblioteki wykorzystującej zrównoleglenie. Jej wykorzystywanie ma sens dla obrazów relatywnie dużych.

Kolejnym przeprowadzony testem był test konstruktora StereoBM(). Wyniki przedstawione są w tabeli 5.3

Seria	1	2	3	4
Rozdzielczość obu obrazów wejściowych	1920x1440	1280x960	640x480	160x120
Średni czas wykonywania sekwencyjnego	3440,8	1102,7	162	6,3
Średni czas wykonywania równoległego	1463,7	480,5	92,8	6,2
Średnie przyspieszenie	2,35	2,29	1,75	1,02

Tablica 5.3: Wyniki testów Stereo Match

Wyniki testu jasno obrazują sytuację podobną do poprzednich testów. Przy obrazie o rozdzielczości 1920x1440 widać wyraźnie skrócenie czasu wykonywania obliczeń. W przypadku takim zysk ze zrównoleglenia wyniósł 2,35. Wynik ten jest jednak wynikiem dosyć przeciętnym. Spadek problemu obliczeniowego przekłada się na spadek przyspieszenia. Narzut potrzebny do rozpoczęcia pracy wielowątkowej przerasta zysk z wykorzystania mechanizmów

współbieżnych. Algorytm został zrównoleglony dosyć poprawnie. Wymaga jednak ostrożnego stosowania, gdyż w pewnych przypadkach może wydłużyć czas wykonywania obliczeń.

Czwartym wykonanym testem był test funkcji CvGBTrees.train(). Otrzymane wyniki prezentują się tak jak w tabeli 5.4.

Seria	1	2	3	4
Liczba elementów w pliku wejściowym	5000	2500	1000	300
Średni czas wykonywania sekwencyjnego	2397,8	1238,8	508,7	149,6
Średni czas wykonywania równoległego	2105,5	1201,9	583,6	233,7
Średnie przyspieszenie	1,14	1,03	0,87	0,64

Tablica 5.4: Wyniki testów GBTrees

Po odczytaniu tabeli od razu widać, że i w tym przypadku wartość przyspieszenia zależna jest od złożoności problemu. Przy pięciu tysiącach elementów widać niewielki zysk z wykorzystania zrównoleglenia, a przy trzystu elementach, można mówić o stratach, a nie zysku. Jakość zrównoleglenia w algorytmie nie jest więc na zbyt wysokim poziomie. Dopiero przy bardzo dużej liczbie elementów zysk staje się widoczny. Wykorzystanie zrównoleglonej wersji algorytmu wymaga więc, aby rozwiązywane były bardziej złożone problemy. W przypadku rozwiązywania prostszych, lepiej wykorzystać wersję sekwencyjną algorytmu.

Ostatnimi testami były testy nauki klasyfikatora MlAnn. Wyniki, tak jak poprzednio, przedstawione zostały w postaci tabeli 5.5.

Seria	1	2	3	4
Liczba elementów do nauki klasyfikatora	900	450	180	90
Średni czas wykonywania sekwencyjnego	516,4	101,1	55,2	19,8
Średni czas wykonywania równoległego	17090,3	2544,6	1518,7	236,3
Średnie przyspieszenie	0,03	0,04	0,04	0,08

Tablica 5.5: Wyniki testów MlAnn

Otrzymane wyniki były trochę zaskakujące. Okazało się, że wraz ze wzrostem problemu obliczeniowego spada też przyspieszenie. Pokazuje to, że algorytm nie został prawidłowo zrównoleglony. Prawdopodobnie pracujące wątki wzajemnie się wykluczają i blokują powodując w ten sposób niepotrzebne oczekiwanie. Wszystko to przekłada się na straty z powodu wykorzystania mechanizmów zrównoleglających. W tym przypadku stanowczo lepiej wykorzystywać jest bibliotekę nie wykorzystującą Intel TBB.

5.2 Testy algorytmów własnych

Zadania testowe do programu drugiego zostały rozbite na dwa typy. Pierwsze z nich różniły się wielkością obrazu wejściowego przy stałej liczbie wątków. Drugie różniły się ilością wykorzystywanych wątków przy stałej wielkości obrazu wejściowego. Każdy z testów przeprowadzony był pięciokrotnie. Zmniejszenie liczby testów spowodowane było otrzymywaniem w miarę podobnych rezultatów przy wykonywaniu testów programu pierwszego. Stwierdzono, że dziesięciokrotne powtarzanie testu daje taki sam rezultat co jego pięciokrotne wykonanie.

W tabeli 5.6 przedstawione są wyniki własnego algorytmu zrównoleglającego funkcję SurfFeaturesDescriptor.detect().

Seria	1	2	3	4
rozdzielczość obrazu wejściowego	1632x1224	816x612	408x306	102x77
Średni czas wykonywania sekwencyjnego	2376,6	625,2	173	11,2
Średni czas wykonywania równoległego	1542,2	375,2	96,6	4,8
Średnie przyspieszenie	1,54	1,67	1,79	2,3

Tablica 5.6: Wyniki testów własnego zrównoleglenia funkcji SurfFeaturesDescriptor.detect()

Wyniki pokazują, że zrównoleglenie się udało, lecz jego efekty nie są najlepsze. W czasie tworzenia algorytmu przyjęta została strategia cięcia obrazu na mniejsze obrazy w postaci poziomych pasów. Podejście takie wymaga jednak, aby pasy te się zazębiały, gdyż algorytm detekcji punktów kluczowych automatycznie odrzuca punktu znajdujące się zbyt blisko krawędzi obrazu. W przeciwnym wypadku, odnalezionych punktów kluczowych na mniejszych

obrazach będzie po prostu mniej, a co za tym idzie rezultaty obu algorytmów byłyby inne. Jednak dodanie zazębiania zwiększa liczbę pikseli, które należy przetworzyć. Dlatego właśnie zysk z własnoręcznego zrównoleglenia algorytmu dla czterech wątków nie przekroczył wartości 2. Ponadto widać, że algorytm lepiej działa dla mniejszych obrazów. Oznacza to dokładnie tyle, że konieczność dodania pasów znacznie ogranicza możliwość zrównoleglania algorytmu.

Kolejnym testowanym algorytmem był algorytm dokonywania erozji obrazu. Dla porównania wykonany został jeszcze test wykonywania erozji przy pomocy jednej z funkcji biblioteki OpenCV. Średnie wyniki widać w tabeli 5.7

Seria	1	2	3	4
rozdzielczość obrazu wejściowego	1632x1224	816x612	408x306	102x77
Średni czas wykonywania sekwencyjnego	1142,2	282,8	67,8	2,2
Średni czas wykonywania równoległego	555,4	130	30,8	2,0
Średni czas wykonywania funckji biblioteki OpenCV	21,8	7,2	2	0,6
Średnie przyspieszenie	2,06	2,18	2,2	1,1

Tablica 5.7: Wyniki testów zrównoleglenia własnej funckji dokonującej erozji obrazu

Tym razem udało się uzyskać dosyć przyzwoite wyniki. Algorytm dokonywania erozji obrazu dał się dobrze zrównoleglić. Tak samo jak w przypadku natywnie zrównolelgonych algorytmów biblioteki OpenCV, jakość zrównoleglenia zależna jest od wielkości obrazu wejściowego. Gdy obraz jest bardzo mały, to zysk ze zrównoleglenia jest niewielki. Wraz ze wzrostem wielkości obrazu, rośnie też zysk ze zrównoleglenia.

Wyniki wykorzystujące bibliotekę OpenCV są przynajmniej o rząd niższe od algorytmów własnych. Wersja funkcji dokonującej erozji wbudowana w bibliotekę jest o wiele wydajniejsza. Test ten pokazuje, że nie warto pisać własnych funkcji, gdy możemy użyć funkcji bibliotecznej. Optymalizacja w nich zawarta jest tak duża, że przerasta nawet zysk jaki możemy otrzymać wykorzystując więcej niż jeden rdzeń procesora.

Trzecim i ostatnim testem był test własnego algorytmu liczenia momentów dla obrazu binarnego. Uzyskane wyniki przedstawione są w tabeli 5.8.

Seria	1	2	3	4
rozdzielczość obrazu wejściowego	1608x1600	804x800	402x400	201x200
Średni czas wykonywania sekwencyjnego	390,4	107	26,2	6,4
Średni czas wykonywania równoległego	165,4	41,4	10	3,4
Średnie przyspieszenie	2,36	2,58	2,62	1,88

Tablica 5.8: Wyniki testów zrównoleglenia własnej funckji liczącej momenty dla obrazu wejściowego

Jak widać po wynikach, zysk ze zrównoleglenia jest tutaj największy. Algorytm ten dało się dobrze zrównolelgić. Praktycznie dla każdej wielkości obrazu zysk ze zrównoleglenia wyniósł ponad 2. Można też zauważyć, że przyspieszenie osiągnęło największy współczynnik dla rozdzielczości obrazu 402x400, a potem zaczęło maleć. Największy wpływ na wyniki miało wykorzystanie w algorytmie wielowątkowym mechanizmu zamka. Bez wykorzystania zamka, zysk ze zrównoleglenia sięgał wartości 3,2. Jednak zauważono, że obliczone momenty w przypadku sekwencyjnym i równoległym były inne. Po jego dodaniu, zysk ze zrównoleglenia spadł, ale wartości uzyskanych momentów były identyczne.

Poza testami z różnymi wielkościami obrazów, dokonane zostały testy z różną liczbą wątków. Pomimo, iż maszyna posiadała tylko cztery fizyczne rdzenie, to ze względu na zastosowaną w procesorze technologię HyperThreading[1] podjęto próby uruchomienia programu z jeszcze większą liczbą wątków. Otrzymane wyniki przedstawia tabela 5.9

Można zauważyć, ze wraz ze wzrostem liczby rdzeni, wzrasta współczynnik przyspieszenia. Najszybciej rośnie on dla algorytmu liczenia erozji obrazu, najwolniej dla zrównoleglenia algorytmu funkcji SurfFeaturesDetector.detect(). W żadnym przypadku wzrost ten nie jest liniowy. Największa różnica w wielości otrzymanego zysku widoczna jest pomiędzy wykorzystaniem dwóch i trzech rdzeni. Dokładanie kolejnych rdzeni daje coraz mniejszy zysk. Zgodne to jest z przedstawionym wcześniej prawem Ahmdal'a. Ponadto

[1]technologia wprowadzona przez firmę Intel, umożliwiająca jednoczesne wykonywanie dwóch wątków na raz przez jeden fizyczny rdzeń procesora, instrukcje z każdego wątku są układane w ten sposób by procesor mógł je wykonywać możliwie najefektywniej (źródło [11])

algorytm:	SURF	Erozja	Momenty
2 wątki	1,31	1,57	1,46
3 wątki	1,34	1,78	1,57
4 wątki	1,54	2,06	2,01
5 wątków	1,68	2,2	1,81
6 wątków	1,76	2,41	2,1
8 wątków	1,76	2,41	2,1

Tablica 5.9: Wyniki testów z różną liczbą wątków

widać, że technologia HyperThreading pozwala na wykonywanie dodatkowych instrukcji w wolnym czasie, a co za tym idzie, dodatkowym przyspieszeniu wykonywania programu.

Ostatnim przeprowadzonym testem był test Video. Test ten polegał na nałożeniu filtru bilateralnego na obraz pochodzący z kamery. Obraz był kopiowany jednokrotnie, a następnie na oba obrazy nakładany był filtr bilateralny. Sekwencyjna wersja algorytmu nakładała filtr kolejno, najpierw na jeden obraz, potem na drugi. Zrównoleglona wersja algorytmu dokonywała filtracji obu obrazów niezależnie. Tabela 5.10 przedstawia wyniki testu video.

algorytm	FPS min	FPS max	FPS śr.
sekwencyjny	3,76	4,41	4,06
dwuwątkowy	6,89	7,54	7,25

Tablica 5.10: Wyniki testów z różną liczbą wątków

Jak widać średnia liczba klatek na sekundę różni się znacznie w obu przypadkach. Dla algorytmu sekwencyjnego waha się w okolicach czterech klatek na sekundę. Równoległy algorytm wyświetlał obraz z prawie dwukrotnie większą liczbą klatek na sekundę. Średnie przyspieszenie z zastosowania zrównoleglenia wyniosło więc 1,79. Wynik ten jest dobry, jeśli wziąć pod uwagę, że program wykorzystywał tylko dwa rdzenie procesora.

Zaprezentowane podejście umożliwiło przetwarzanie obrazu w czasie rzeczywistym. Do kodu sekwencyjnego wystarczyło dodać dwie linie oraz dwie dyrektywy dla kompilatora umożliwiającą wykorzystywanie Intel TBB. Taki prosty zabieg umożliwił prawie dwukrotny wzrost liczby wyświetlanych klatek na sekundę. Przykład ten pokazuje, jak proste zabiegi mogą doprowadzić to znacznego wzrostu wydajności aplikacji. Przy niewielkiej modyfikacji programu można uzyskać dosyć interesujące rezultaty.

5.3 Podsumowanie

Wykonanie wszystkich testów oraz uśrednienie wyników dało obraz zrównoleglenia w poszczególnych algorytmach. Dzięki takiemu zestawieniu można w szybki i prosty sposób porównać jakość zrównoleglenia w poszczególnych algorytmach. Zestawienie wszystkich wyników widoczne jest w tabeli 5.11

Test	S max	S min	S śr
Surf	2,23	1,87	2,12
Cascade Classifier	2,67	0,94	1,93
Stereo Match	2,35	1,02	1,85
GBTrees	1,14	0,64	0,92
MlAnn	0,08	0,03	0,05
Surf wlasne	2,3	1,54	1,83
Erozja	2,2	1,1	1,89
Momenty	2,62	1,88	2,36

Tablica 5.11: Średnie przyspieszenie dla każdego z testów

Największe przyspieszenie w pojedynczym zadaniu osiągnął test obliczania momentów. Wynik tego testu jest również najlepszy pod względem średniego przyspieszenia. Najmniejsze przyspieszenie, a właściwie największe straty przyniósł test MlAnn. Świadczy to o złym zrównolegleniu funkcji.

Z tabeli można też odczytać, które algorytmy uzyskują zysk ze zrównoleglenia przy praktycznie każdych danych wejściowych, a które przy małej ilości danych ponoszą straty. Na uwagę zasługuje zrównoleglenie funkcji SurfFeaturesDescriptor.detect(). Widać, że jest ono bardzo stabilne w porównaniu do zrównoleglenia pozostałych funkcji.

Rozdział 6

Wnioski

Przeprowadzona analiza pomogła odpowiedzieć na pytania zadane przed rozpoczęciem pracy nad projektem. Udało się zrealizować wszystkie cele, które zostały postawione we wstępie. Stworzono projekt w MS Visual Studio umożliwiający kompilację programu w dwóch wersjach. Pierwsza z nich wykorzystuje bibliotekę OpenCV z Intel TBB, druga z nich używa OpenCV bez Intel TBB. W źródłach biblioteki OpenCV odnalezione zostały funkcje, które mają zaimplementowane zrównoleglenie. Do pięciu odnalezionych funkcji napisane zostały odpowiednie testy. Wyniki testów mierzone były przy pomocy napisanej wcześniej klasy, a następnie zapisane. Następnie napisany został własny algorytm wykorzystujących Intel TBB wraz z OpenCV oraz dwa algorytmy wykorzystujących tylko Intel TBB. Te algorytmy również zostały przetestowane, a wyniki zapisane.

Wykonane testy ukazały jak wielowątkowość może zostać użyta przy przetwarzaniu obrazów. Przeprowadzone doświadczenia pomogły sprawdzić w jakim stopniu mechanizmy wbudowane w bibliotekę OpenCV mogą przyspieszyć obliczenia. Dodatkowo, projekt spowodował, że poznałem lepiej budowę zarówno biblioteki OpenCV jak i Intel TBB. Wykorzystanie ich funkcjonalności wydaje się być łatwiejsze.

6.1 Dlaczego IntelTBB ?

Twórcy biblioteki OpenCV pierwotnie korzystali z dobrodziejstw innej otwartej biblioteki, a dokładniej OpenMP. To właśnie dzięki niej, zrównoleglane były przeróżne funkcje. Jednak od jakiegoś czasu zrównoleglanie wykonywane jest przy pomocy biblioteki Intel TBB. Poprzednie mechanizmy zostały zupełnie usunięte. Ciężko jednoznacznie powiedzieć dlaczego autorzy zdecydowali się na taki krok. Co prawda praca nie miała na celu skupiać się

na porównani wydajności obu bibliotek, jednak podczas szukania informacji na temat Intel TBB odnalezione zostało porównanie [10]. Wynikało z niego jasno, że stosowanie biblioteki Intel TBB zamiast OpenMP jest po prostu prostsze, a często nawet wydajniejsze.

W bibliotece Intel TBB użytkownik jest zwolniony z obowiązku skalowania problemu dającego się zrównolelgić pomiędzy rdzenie procesora. Jedynym wymogiem jest zapoznanie się z budową funkcji czy pętli, którą chce się zrównoleglić i odpowiednie jej zastosowanie. Z początku sprawiało mi to pewne kłopoty, gdyż sama budowa funkcji wydawała się dziwna i trochę nieintuicyjna. Pod koniec tworzenia programów zdałem sobie jednak sprawę, że tak naprawdę nie jest to aż takie skomplikowane. Mało tego, okazało się, że faktycznie nie trzeba martwić się o zasoby i przydzielanie ich odpowiednim rdzeniom.

Podczas testów nie miałem możliwości sprawdzenia programów na komputerach z innymi procesorami, dlatego nie jestem w stanie powiedzieć czy faktycznie biblioteka Intel TBB jest przenośna. Wydaje mi się jednak, że jest ona przenośna i gdybym tylko uruchomił utworzone przeze mnie programy na innych maszynach o wielu rdzeniach, to prawdopodobnie wykorzystywałyby one również potencjał danego sprzętu.

6.2 Wnioski z przeprowadzonej analizy

Analizując zrównoleglenia wbudownae w bibliotekę OpenCV okazało się, że nie jest ono zbytnio rozbudowane. Mimo teoretycznie dużej możliwości wykorzystania wielu wątków przy przetwarzaniu obrazów, autorzy zrównoleglili bardzo małą liczbę funkcji. Prawdopodobnie, skupiają się oni na optymalizacji algorytmów i ich rozbudowie. Kwestie zrównoleglenia pozostawiają raczej programistom wykorzystującym bibliotekę.

Z przetestowanych pięciu funkcji w trzech z nich udało się zauważyć wyraźny wzrost wydajności przy zastosowaniu biblioteki wykorzystującej Intel TBB. Wzrost ten przy czterech rdzeniach oscylował w granicach dwóch razy. Współczynnik przyspieszenia zależał bardzo od wielkości przetwarzanych danych. Im większy obraz wejściowy, tym większy zysk ze zrównolelgenia. Algorytmy wykonywane sekwencyjnie i wielowątkowo dawały identyczne wyniki jeśli chodzi o obraz wyjściowy. Wystarczy podmienić bibliotekę OpenCV działającą sekwencyjnie na taką z obsługą Intel TBB i programy, które wykorzystują jedną z trzech wspomnianych wcześniej funkcji najzwyczajniej zaczną działać szybciej. Trzeba jednak pamiętać, że zysk ten będzie zależny od wielkości przetwarzanych obrazów.

Z drugiej strony, znalazły się też dwa algorytmy zrównoleglone słabiej.

Jeden z nich nawet przy dużej liczbie danych wejściowych działał niewiele szybciej. Drugi zaś, działał po prostu gorzej w wersji zrównoleglonej niż w wersji sekwencyjnej. Jak widać nie wszystko zostało napisane poprawnie. Dlatego, jeśli w programie używana jest któraś z tych funkcji to w zasadzie można pominąć używanie zrównoleglonej wersji biblioteki.

Biorąc pod uwagę testy z programu drugiego, można dojść do wniosku, że wiele algorytmów dałoby się zrównoleglić nawet na kilka sposobów. Pozostaje tylko pytanie, czy jest to opłacalne ? Programy przetwarzające obrazy przy pomocy biblioteki OpenCV działają na różnych danych wejściowych. W niektórych przypadkach zysk ze zrównoleglenia mógłby być całkiem spory, w innych mniejszy lub zerowy. Dlatego właśnie zrównolelganie zostało zostawione użytkownikom. Od ich decyzji i sposobu, w który wykorzystują bibliotekę zależy czy warto wykorzystać potencjał kilku rdzeni.

W czasie wykonywania testów udało się też potwierdzić prawo Ahmdala. Wraz ze wzrostem liczby wykorzystywanych do zrównoleglenia procesorów, wzrasta przyspieszenie. Jednak przyspieszenie to nie wzrasta liniowo. Największe znaczenie ma tutaj część kodu, która musi działać sekwencyjnie, bo nie da się jej zrównoleglić. Dlatego właśnie, dołożenie czwartego wątku spowodowało mniejszy wzrost przyspieszenia niż dołożenie wątku trzeciego.

6.3 Czy w bibliotece warto coś zmienić ?

Odpowiedź na to pytanie nie jest jednoznaczna. Wydaje się iż większość funkcji, która mogłaby być dosyć łatwo zrównoleglona, jednak zrównoleglana nie jest, gdyż głównym priorytetem twórców biblioteki jest rozszerzanie spektrum jej działania poprzez dodawanie nowej funkcjonalności, a nie wielowątkowa optymalizacja istniejącego kodu. Wynika to, jak sądzę, w dużej mierze z braku odpowiednich mocy przerobowych oraz przyjęcia założenia, że łatwiej jest użytkownikom zrównoleglić istniejący kod, niż zaimplementować zupełnie nowe rozwiązania.

Warto natomiast dodać, że jedna z funkcji jest źle zrównoleglona i powoduje spadek wydajności. Przydałoby się ją poprawić lub usunąć z niej zrównoleglenie. Pozostawienie jej w takim stanie może prowadzić do paradoksalnej sytuacji, kiedy użytkownik chcąc przyspieszyć obliczenia, po prostu je spowolni.

6.4 Możliwości rozbudowy

Wykonana praca inżynierska nie rozwinęła wszystkich możliwych kwestii. Skupiała się ona na najbardziej znaczących elementach przetwarzania wielowątkowego w bibliotece OpenCV. Przetestowane zostały prawie wszystkie zrównoleglone w bibliotece funkcje. W tej kwestii nie ma już raczej zbyt wiele do zrobienia. Można oczywiście przetestować pozostałe funkcje, ale prawdopodobnie rezultaty testów będą podobne do tych wykonanych w czasie pisania pracy.

W ramach rozbudowy pracy, warto byłoby jednak spróbować własnoręcznie zrównoleglić więcej funkcji. Najważniejsze, by próby te pokazały, które funkcje da się łatwo zrównoleglić, a które lepiej pozostawić w spokoju. Dobrze byłoby też sprawdzić jak zachowują się różne sposoby zrównoleglania. Przykładowo, czy lepiej dzielić obrazy na poziome pasy, czy też może na pasy pionowe, a może w ogóle dzielić obraz na składowe koloru. Mnogość algorytmów zawartych w bibliotece, daje naprawdę duże możliwości wykonywania obliczeń równoległych. Możliwych sposobów zrównoleglania tych algorytmów jest jeszcze więcej. Żaden sposób nie jest i nie będzie idealny, ale każdy z nich miałby jakieś zalety i wady.

Z drugiej strony, można by porównać bibliotekę Intel TBB z innymi bibliotekami do tworzenia aplikacji współbieżnych. Przedstawione tutaj algorytmy, można by napisać od nowa wykorzystując zupełnie inną bibliotekę lub natywne mechanizmy wątków z systemu operacyjnego. Otrzymane wyniki prawdopodobnie różniłby się od tych, które zostały tutaj udokumentowane. To, czy byłyby lepsze, czy gorsze, zależałoby zarówno od samych zastosowanych bibliotek, jak i od tego jak radzą sobie one z określonymi problemami.

Bibliografia

[1] Gary Bradski, Adrian Kaehler, „Learning OpenCV: Computer Vision with the OpenCV Library", O'Reilly Media, 2008

[2] Robert Laganière, „OpenCV 2 Computer Vision Application Programming Cookbook" , Packt Publishing, 2011

[3] Ewaryst Rafajłowicz, Wojciech Rafajłowicz, Andrzej Rusiecki, „Algorytmy przetwarzania obrazów i wstęp do pracy z biblioteką OpenCV", Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław 2009

[4] Dokumentacja biblioteki OpenCV, dostępna w internecie: [dostęp 4 stycznia 2012]
`http://opencv.itseez.com/`

[5] Dokumentacja biblioteki Intel TBB, dostępna w internecie: [dostęp 4 stycznia 2012]
`http://threadingbuildingblocks.org/documentation.php`

[6] Strona internetowa firmy Intel, artykuł o zarządcy zadań biblioteki Intel TBB, dostępny w internecie: [dostęp 7 stycznia 2012]
`http://www.intel.com/technology/itj/2007/v11i4/5-foundations/4-tbb.htm`

[7] Multithreading Tutorial, informacje o mechaniźmie zamków, dostępny w internecie: [dostęp 10 stycznia 2012]
`http://www.paulbridger.com/mutexes/`

[8] Practical 4 - Basic OpenMP, Tim Stitt Ph.D.., dostępna w internecie: [dostęp 6 stycznia 2012]
`http://cnx.org/content/m32284/latest/`

[9] prof. nzw. dr hab. inż. Starzyński Jacek, wykład nr 2 z przedmiotu "Programownie równoległe i rozproszone", dostępny w internecie [dostęp 9 stycznia 2012]

http://wikidyd.iem.pw.edu.pl/PRiR?action=AttachFile&do=view&target=wyklad2.pdf

[10] Compare Windows threads, OpenMP, Intel Threading Building Blocks for parallel programming, dostępna w internecie [dostęp 15 stycznia 2012]
http://software.intel.com/en-us/blogs/2008/12/16/compare-windows-threads-openmp-intel-threading-building-blocks-for-parallel-programming/

[11] „Intel 64 and IA-32 Architectures Software Developer's Manual", dostępna w internecie [dostęp 16 stycznia 2012]
http://download.intel.com/design/processor/manuals/253665.pdf

[12] Strona domowa Mariusa Muja, informacje o FLANN, dostępne w internecie [dostęp 16 stycznia 2012]
http://www.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN

[13] Strona domowa Łukasza Piątkiewicza, dostępna w internecie [dostęp 18 stycznia 2012]
http://www.analizaobrazu.friko.pl/teoria.html