

**Zachodniopomorski Uniwersytet
Technologiczny w Szczecinie
Wydział Elektryczny**



Radosław Rajczyk

nr albumu: 23804

**Implementacja algorytmu Viterbiego
z wykorzystaniem biblioteki OpenCL**

Praca dyplomowa magisterska
kierunek: Automatyka i Robotyka
specjalność: Systemy sterowania procesami przemysłowymi

Opiekun pracy:
dr hab. inż. Przemysław Mazurek
Katedra Przetwarzania Sygnałów i Inżynierii Multimedialnej
Wydział Elektryczny

Szczecin, 2016

Spis treści

1	Streszczenie	3
2	Wstęp	4
2.1	Przetwarzanie obrazu i jego rola w automatyce przemysłowej	4
2.2	Istotność szybkości obliczeń w problemach wizji maszynowej	5
2.3	Cel, zakres i zastosowania pracy	7
3	Metody równoległego przetwarzania danych	8
3.1	Wielowątkowość aplikacji dla języka C/C++	8
3.1.1	Biblioteka POSIX dla systemów Unix	9
3.1.2	OpenMP - wieloplatformowe API	12
3.1.3	Wielowątkowość w standardzie C++11	15
3.2	Programowanie równoległe z wykorzystaniem GPU	17
3.2.1	Architektura GPU i porównanie względem CPU	17
3.2.2	Biblioteka OpenCL	17
4	Algorytm Viterbiego	18
4.1	Opis działania i zastosowania	18
4.2	Implementacja w języku C++	18
4.2.1	Wersja szeregową	18
4.2.2	Wersja równoległa - C++11	18
4.2.3	Wersja równoległa - OpenCL	18
5	Wyniki badań doświadczalnych implementacji algorytmu Viterbiego	19
5.1	Porównanie czasu działania dla implementacji szeregowej, wielowątkowej oraz z wykorzystaniem biblioteki OpenCL	19
5.2	Porównanie szybkości algorytmów dla różnych konfiguracji sprzętowych	19
6	Wnioski końcowe	20
7	Załącznik B	21
8	Załącznik A	22
	Spis rysunków	23
9	Bibliografia	25

Rozdział 1

Streszczenie

To jest streszczenie

Rozdział 2

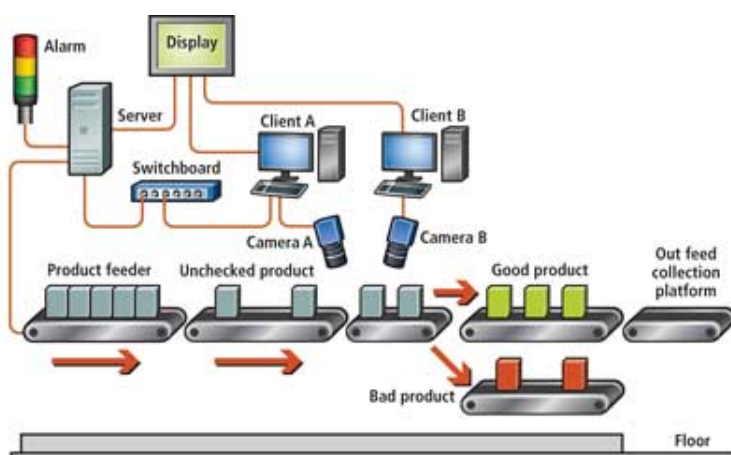
Wstęp

2.1 Przetwarzanie obrazu i jego rola w automatyce przemysłowej

W zagadnieniach technik pomiarowych oraz analizy otoczenia coraz częściej stosowane są rozwiązania wykorzystujące systemy wizyjne. Do najpopularniejszych zastosowań przemysłowych wizji maszynowej należą [12]:

- inspekcja elementów na linii technologicznej
- określanie właściwej orientacji i położenia elementów
- identyfikacja produktów
- pomiary metrologiczne

W automatyce przemysłowej gdzie do zagadnień inspekcji wcześniej niezbędna była ocena wizualna człowieka, obecnie powszechnie stosuje się systemy wizyjne, w których skład wchodzi kamery przemysłowe, czujniki wyzwalające (np. na bazie pozycji) oraz urządzenie odpowiadające za proces decyzyjny. Występują również rozwiązania w postaci systemów wbudowanych, gdzie inteligentna kamera oprócz akwizycji obrazu zajmuje się jego przetwarzaniem i analizą, wykorzystując własny procesor. [12][14]



Rysunek 2.1: Przykład zautomatyzowanej linii technologicznej wykorzystującej system wizyjny[31]

Sprawdzanie orientacji i położenia elementów w przemyśle jest wykorzystywane między innymi w technologii montażu, gdzie informacje z urządzeń wizyjnych są wykorzystywane przez manipulatory przemysłowe do zautomatyzowanego montażu, sortowania oraz paletyzacji wyrobów.[12]



Rysunek 2.2: Przykład obrazów używanych w testowaniu pozycji i orientacji elementów[12]

Identyfikowanie produktów na bazie obrazu cyfrowego jest wykorzystywane przy sortowaniu oraz monitorowaniu przepływu elementów i lokalizacji wąskich gardeł. Przykładowe metody indentyfikacji to stosowanie kodów kreskowych i kodów DataMatrix.[12]



Rysunek 2.3: Przykład wizyjnej identyfikacji[12]

2.2 Istotność szybkości obliczeń w problemach wizji maszynowej

Większość praktycznych zastosowań przetwarzania obrazu jako dodatkowej informacji w sterowaniu jednym bądź grupą urządzeń, wymaga akwizycji oraz wykonywania obliczeń w czasie rzeczywistym. Oznacza to, że wybrany algorytm wykorzystywany do analizy obrazu cyfrowego, wraz z resztą niezbędnego kodu, musi posiadać czas wykonania spełniający narzucone przez sterowany system.

Dla zastosowań przemysłowych, gdzie monitorowane obiekty poruszają się z dużą prędkością, szybkość podjęcia decyzji przez system wizyjny może być wąskim gardłem dla danej gałęzi linii produkcyjnej. Czas na wykonanie decyzji (np. o usunięciu wadliwego produktu z przenośnika taśmowego), składa się na czas akwizycji obrazu, obliczenia sterowania. Standardowe kamery przemysłowe potrafią zrobić nawet powyżej 100 zdjęć na sekundę, a nawet więcej stosując mniejsze rozdzielczości obrazu. Czas przesyłu danych dla standardu popularnego standardu GigE wynosi maksymalnie 125 MB/s [19]. Na podstawie tego można stwierdzić, że główny problem będzie stanowił czas obliczenia sterowania i od niego będzie zależeć szybkość działania systemu wizyjnego[11][7].

Inną dziedziną gdzie stosowane jest przetwarzanie obrazu w czasie rzeczywistym jest robotyka mobilna, gdzie system wizyjny może odpowiadać za:

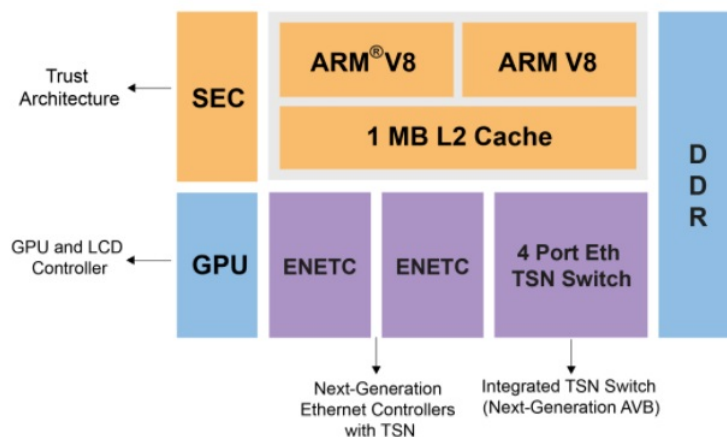
- Sprzężenie niezbędne do obliczenia zmiany położenia i prędkości robota.
- Lokalizację przeszkód oraz innych robotów (Swarm Robotics).
- Analizę oraz monitorowanie otoczenia.

[18][1]

Podobnie jak dla aplikacji przemysłowych odpowiedzialność za wykorzystanie pełnych możliwości układów wykonawczych robota jest szybkość obliczania nowych sterowań. Niezbędne obliczenia mogą być wykonywane bezpośrednio przez urządzenie sterujące silnikami robota, albo z pomocą osobnej stacji, która połączona zdalnie z kontrolerem robota jest odpowiedzialna za przeprowadzanie czasochłonnych obliczeń.

Pierwsze rozwiązanie jest korzystne kiedy nie są wymagane duże rozdzielczości obrazu, skomplikowane i czasochłonne algorytmy przetwarzania obrazu o dużej złożoności obliczeniowej oraz wysokie prędkości ruchu robota, narzucające krótki czas na obliczenia. Stosowane są wtedy najczęściej układy wyposażone w mikroprocesory, np. rodzina procesorów ARM oraz x86-64 firmy Intel. Obecne modele są najczęściej wielordzeniowe o taktowaniu nawet powyżej 1GHz [27] [20].

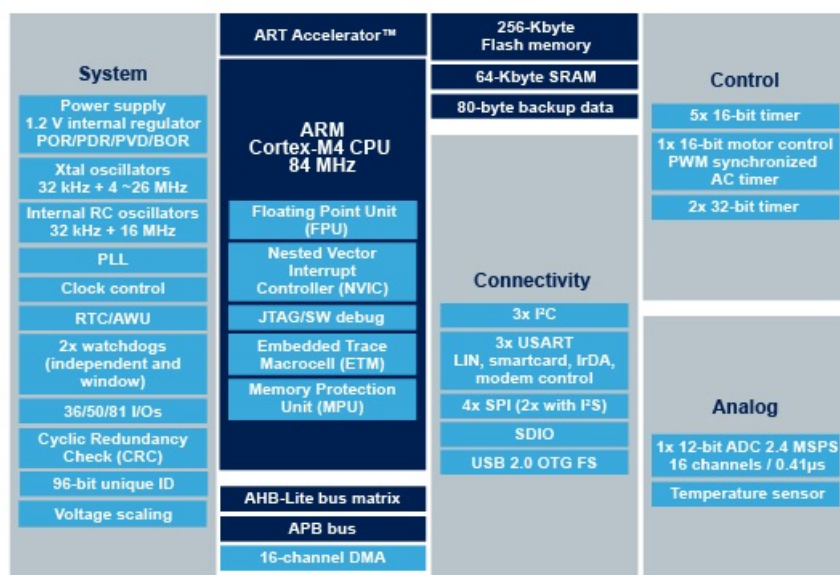
QorIQ® Layerscape LS1028 A Block Diagram



Rysunek 2.4: Procesor *QorIQ Layerscape LS1028* do aplikacji przemysłowych firmy NXP, wyposażony w dwa rdzenie ARMv8[27]

Dla rozwiązań bardziej wymagających pod względem szybkości obliczeń stosowane są układy FPGA, w których logika przetwarzania informacji obrazu jest zaprojektowana w języku HDL (VHDL, Verilog). Zapewniają one najszybsze prędkości obliczeń ze względu na sprzętową implementację algorytmów. [17][10].

Drugie rozwiązanie gdzie osobne urządzenie jest używane do przetwarzania obrazu, umożliwia użycie mniej kosztownego procesora po stronie robota. Wystarczające do sterowania silnikami jest zastosowanie układu wykorzystującego mikrokontroler (np. z rodziny STM32 bądź Atmel AVR) [29] [3].



Rysunek 2.5: Structura mikrokontrolera *STM32F401CC*[29]

Zewnętrzna jednostka obliczeniowa pozwala na wykorzystanie możliwości konwencjonalnych wielordzeniowych procesorów dla komputerów PC oraz procesora karty graficznej, który jest wyspecjalizowany w obliczeniach równoległych[26] [21][22]. Dzięki kombinacji CPU i GPU możliwe jest przyspieszenie operacji, które mogą być wykonywane równolegle oraz rozdzielenie obciążenia obliczeniowego pomiędzy procesor i kartę graficzną.

Podsumowując, szybkość obliczeń systemu wizyjnego w robotyce mobilnej decyduje o tym jakie mogą być maksymalne parametry ruchu robota - prędkość, przyspieszenie, ilości robotów współpracujących w zagadnieniach robotyki roju(Swarm Robotics) oraz poziomie skomplikowania analizy obrazu. W przypadku problemów wizji maszynowej dla zastosowań przemysłowych czas poświęcony na analizę każdego zdjęcia ma wpływ na szybkość działania całej linii produkcyjnej, co ma bezpośredni wpływ na wydajność i koszty produkcji.

2.3 Cel, zakres i zastosowania pracy

Celem pracy jest implementacja algorytmu Viterbiego w celu wykrywania linii na zaszumionym obrazie cyfrowym oraz analiza porównawcza dla różnych wersji napisanego algorytmu. Rozpatrywana będzie implementacja szeregową i równoległą dla CPU w języku C++ oraz napisana pod procesor karty graficznej z wykorzystaniem biblioteki OpenCL. Implementacja z wykorzystaniem biblioteki OpenCL będzie składała się z dwóch wariantów:

- całkowicie wykonywany przez GPU
- hybrydowy - podział obciążenia obliczeniowego pomiędzy procesor i kartę graficzną.

Następnie dla różnych konfiguracji sprzętowych zostanie zrobione porównanie ich szybkości. Na podstawie powyższej analizy zostanie wybrany najlepszy wariant realizacji algorytmu Viterbiego, co będzie mogło być później zastosowane w sterowaniu ruchem robota mobilnego.

Rozdział 3

Metody równoległego przetwarzania danych

3.1 Wielowątkowość aplikacji dla języka C/C++

Najbardziej popularnym podejściem do pisania aplikacji jest sekwencyjne wykonywanie instrukcji przez procesor - tylko jedna z nich może być wykonywana w tym samym czasie. Jednak dużą ilość problemów można rozbić na niezależne fragmenty, które mogą być rozwiązywane równolegle. Obecnie powszechnie stosowane procesory wielordzeniowe dają możliwość rozdzielenia obciążenia obliczeniowego na poszczególne rdzenie oraz dla każdego rdzenia na osobne wątki.[21][4]

Wątek jest to podproces, który posiada własny stos, zestaw rejestrów, ID, priorytet i wykonuje określony fragment kodu programu. W przeciwieństwie do prawdziwego procesu posiada wspólną pamięć globalną i sterty, dzieloną z innymi wątkami istniejącymi w ramach tego samego procesu. Wątki procesu wykonują się równolegle, dopóki nie potrzebują dostępu do zasobów we wspólnej pamięci.[6][23] Wtedy ze względu na problem błędnego odczytu, bądź zapisu wartości w pamięci kiedy inny wątek ją już nadpisał, może spowodować niewłaściwe działanie programu. Fragmenty kodu gdzie może dojść do tego problemu nazywane są sekcjami krytycznymi. Do zabezpieczania sekcji krytycznych programu stosowane są blokady - muteksy (Mutual exclusions) oraz zmienne warunkowe. Zastosowanie muteksa powoduje, że w danym momencie tylko jeden wątek może wykonywać kod chroniony przez tą blokadę i dopóki nie opuści chronionej sekcji krytycznej inny wątek nie może zacząć jej wykonywać. Zmienne warunkowe stosowane są do sygnalizowania postępu danego wątku, tak aby inny mógł kontynuować wykonywanie operacji sekcji krytycznej. Stosowane razem z blokadami umożliwiają właściwą synchronizację pracy wątków tego samego procesu. [9][30]

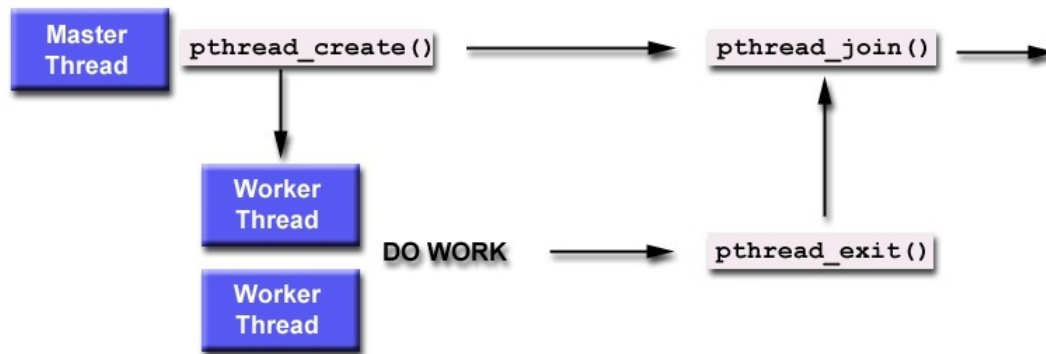
Używanie wielowątkowości w aplikacjach pozwala na wykorzystanie możliwości sprzętowych procesorów wielordzeniowych do obliczeń równoległych. Ponadto wielowątkowy model programowania umożliwia wykonywanie przez proces dalszych działań w czasie czasochłonnych obliczeń, bądź czekania na zdarzenie blokujące - takie jak np. sygnał z urządzenia peryferyjnego. Wadą tworzenia dodatkowych wątków w programie jest narzut obliczeniowy związany z ich synchronizacją (omawiany wcześniej problem wyścigu oraz zjawisko zakleszczenia - wątki czekają na siebie nawzajem żeby móc kontynuować obliczenia) oraz dostępem do wspólnego obszaru pamięci.[9][30]

W tym rozdziale zostaną omówione różne metody tworzenia aplikacji wielowątkowych w języku C/C++. Skupiono się na bibliotekach dla tych języków programowania ze względu na ich popularność w tworzeniu rozwiązań dla systemów wbudowanych, która wynika z wysokiej wydajności kodu i małego zużycia pamięci w porównaniu do języków interpretowanych, np. Java, Python.[30][24]

3.1.1 Biblioteka POSIX dla systemów Unix

Dla systemów z rodziny Unix w 1995 ustalony został standard programowania wielowątkowego nazywany POSIX threads, w skrócie Pthreads. API Pthreads zostało zdefiniowane jako zestaw typów i procedur w języku C, zawarte w pliku nagłówkowym `<pthread.h>` i bibliotece `libpthread`. [6] Korzystanie z biblioteki Pthreads do tworzenia wątków generuje mniejszy narzut niż tworzenie osobnych procesów do równoległego przetwarzania danych. [6]

Wątek w programie jest reprezentowany poprzez zmienną typu `pthread_t`, najczęściej zdefiniowaną jako zmienna statyczna lub jako struktura, która jest zaalokowana na stacku. [9][6][23] Do każdego stworzonego wątku przypisana jest funkcja, którą będzie wykonywał. Funkcja powinna przyjmować jako argument zmienną wskaźnikową `void*` i zwracać wartość tego samego typu. Za tworzenie nowego wątku odpowiedzialna jest funkcja `pthread_create`. Przyjmuje ona adres funkcji oraz argument z jakim ma zostać wywołana. Wywołanie `pthread_create` oprócz rozpoczęcia nowego wątku zwraca identyfikator `pthread_t`, który będzie wykorzystywany do odnoszenia się do stworzonego wątku. Wątek zostaje zakończony jeśli wykona wszystkie instrukcje swojej funkcji lub jeśli wywoła procedurę `pthread_exit`. [9]



Rysunek 3.1: Wykorzystanie `pthread_join` do synchronizacji wątków [6]

Jedną z podstawowych metod synchronizacji pomiędzy wątkami jest użycie funkcji `pthread_join`, która powoduje zatrzymanie dalszego wykonywania instrukcji dopóki stworzony wątek nie zakończy pracy. Tylko wątki, które zostały stworzone z atrybutem `joinable`, a nie `detached` (odłączony) mogą używać tego rodzaju synchronizacji. [6] (patrz 3.1).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4
5 void *print_message_function( void *ptr )
6 {
7     char *message;
8     //casts void* to readable char* message
9     message = (char *)ptr;
10    int i;
11    for(i = 0; i < 5; i++)
12    {
13        printf("%s = %d\n", message, i);
14        sleep(1);
15    }
16 }
17
18 int main()
```

```

19 {
20     pthread_t thread1, thread2;
21     const char *message1 = "Thread 1";
22     const char *message2 = "Thread 2";
23     int  iret1, iret2;
24     //create and execute thread1
25     iret1 = pthread_create(&thread1, NULL, print_message_function, (void*)message1);
26     //check if thread1 successfull created
27     if(iret1)
28     {
29         fprintf(stderr, "Error - pthread_create() return code: %d\n", iret1);
30         return iret1;
31     }
32     //create and execute thread2
33     iret2 = pthread_create(&thread2, NULL, print_message_function, (void*)message2);
34     //check if thread2 successfull created
35     if(iret2)
36     {
37         fprintf(stderr, "Error - pthread_create() return code: %d\n", iret2);
38         return iret2;
39     }
40     printf("pthread_create() for thread 1 returns: %d\n", iret1);
41     printf("pthread_create() for thread 2 returns: %d\n", iret2);
42     //wait for threads to finish
43     pthread_join( thread1, NULL);
44     pthread_join( thread2, NULL);
45
46     return 0;
47 }

```

Listing 3.1: Przykład tworzenia i uruchamiania wątków

W celu zapewnienia bezpieczeństwa w współdzieleniu zasobów pomiędzy wątkami podczas wykonywania sekcji krytycznych, najpopularniejsze jest wykluczenie jednoczesnego czytania bądź zapisu wartości w dzielonej pamięci. Używane do tego są zmienne wzajemnego wykluczenia - w skrócie muteks (mutual exclusion). Muteks jest szczególnym przypadkiem semaforu Dijkstry - semaforem binarnym o zbiorze wartości 0, 1.[9][23][6] W bibliotece POSIX threads muteks jest reprezentowany jako zmienna typu `pthread_mutex_t`. W celu posiadania globalnego zasięgu deklarowana jest jako zmienna `static` lub `extern`. [9][24] (patrz 3.2) W celu deklaracji muteksa wykorzystywane jest makro `PTHREAD_MUTEX_INITIALIZER` (patrz . Jeśli muteks jest używany jako element dynamicznie alokowanej struktury, musi zostać zainicjalizowany wywołaniem funkcji `pthread_mutex_init`. Ponadto musi być w ten sposób inicjalizowany, jeśli nie ma posiadać domyślnych atrybutów. Niezbędne jest po zakończeniu używania muteksa, zwolnienie zaalokowanej pamięci, poprzez wykorzystanie funkcji `pthread_mutex_destroy`. [9]

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4
5 pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
6
7 void *protected_fun(void *ptr)
8 {
9     char *message;
10    pthread_mutex_lock( &mutex1 );
11    //casts void* to readable char* message
12    message = (char *)ptr;
13    int i;
14    for(i = 0; i < 5; i++)

```

```

15     {
16         printf("%s = %d\n", message, i);
17         sleep(1);
18     }
19     pthread_mutex_unlock( &mutex1 );
20 }
21
22 int main()
23 {
24
25     pthread_t thread1, thread2;
26     const char *message1 = "Thread 1";
27     const char *message2 = "Thread 2";
28     int  iret1, iret2;
29
30     //create and execute thread1
31     iret1 = pthread_create(&thread1, NULL, protected_fun, (void*)message1);
32     //check if thread1 successfull created
33     if(iret1)
34     {
35         fprintf(stderr, "Error - pthread_create() return code: %d\n", iret1);
36         return iret1;
37     }
38
39     //create and execute thread2
40     iret2 = pthread_create(&thread2, NULL, protected_fun, (void*)message2);
41     //check if thread2 successfull created
42     if(iret2)
43     {
44         fprintf(stderr, "Error - pthread_create() return code: %d\n", iret2);
45         return iret2;
46     }
47
48     printf("pthread_create() for thread 1 returns: %d\n", iret1);
49     printf("pthread_create() for thread 2 returns: %d\n", iret2);
50
51     //wait for threads to finish
52     pthread_join( thread1, NULL);
53     pthread_join( thread2, NULL);
54
55     return 0;
56 }

```

Listing 3.2: Przykład wykorzystanie muteksa do synchronizacji aplikacji wielowątkowej

Pthreads do komunikacji pomiędzy wątkami wykorzystuje zmienne warunkowe(condition variables), które mają informować o stanie współdzielonych zasobów. Używane razem z muteksami, w atomiczny sposób zwalniają blokadę sekcji krytycznej, dopóki inny wątek nie zasygnalizuje kontynuacji używając funkcji `pthread_cond_signal`. Dzięki temu inny wątek może kontynuować pracę zanim zostanie wykonana chroniona sekcja krytyczna.

Dzięki przedstawionym metodom sygnalizacji stanu oraz blokadom, możliwe jest zaprojektowanie pożądanego podziału obciążenia obliczeniowego. Biblioteka POSIX, choć wiekowa dalej jest stosowana dzięki małemu narzutowi(napisana w języku C), obszernej dokumentacji oraz dużej ilości starego kodu, który dalej jest stosowany w nowych rozwiązaniach systemów wbudowanych i czasu rzeczywistego.[23]

3.1.2 OpenMP - wieloplatformowe API

Podobnie jak Pthreads, OpenMP jest biblioteką wykorzystującą model współdzielenia pamięci do programowania równoległego. Zawiera wsparcie dla języków C, C++ oraz Fortran. OpenMP wymaga sprecyzowania przez użytkownika odpowiednich akcji, które ma wykonać kompilator, aby program wykonywał się równolegle. [28][8] OpenMP został stworzony przez grupę naukowców i programistów, którzy uważali, że używanie API Pthreads jest skomplikowane dla dużych aplikacji. Zdecydowali się stworzyć standard wyższego poziomu, który w przeciwieństwie do biblioteki Pthreads wymagającej od programisty zdefiniowania funkcji wykonawczej dla każdego wątku, pozwala na określenie dowolnego fragmentu programu, który ma być wykonany równolegle. Wykorzystuje do tego dyrektywy preprocesora znane jako **#pragma**. Używane są do zdefiniowania zachowań kompilatora, które nie są zawarte w podstawowej specyfikacji języka C. Jeśli użyty kompilator nie wspiera dyrektyw **#pragma**, program i tak ma możliwość właściwego działania; wtedy jego fragmenty mające wykonać się równolegle zostaną obsługiwane przez jeden wątek. [28][24][8][32] Oprócz zestawu dyrektyw preprocesora, OpenMP składa się z biblioteki funkcji i makr, które wymagają dodania pliku nagłówkowego `<omp.h>` z ich definicjami i prototypami.

Podstawową dyrektywą OpenMP jest dyrektywa **#pragma omp parallel**, za pomocą której określany jest blok kodu mający być wykonany wielowątkowo. Jeśli programista nie sprecyzuje przez ile wątków ma być przetworzony podany fragment programu, zostanie on określony przez system wykonawczy (zazwyczaj po jednym wątku na rdzeń). [28][32] (patrz 3.3)

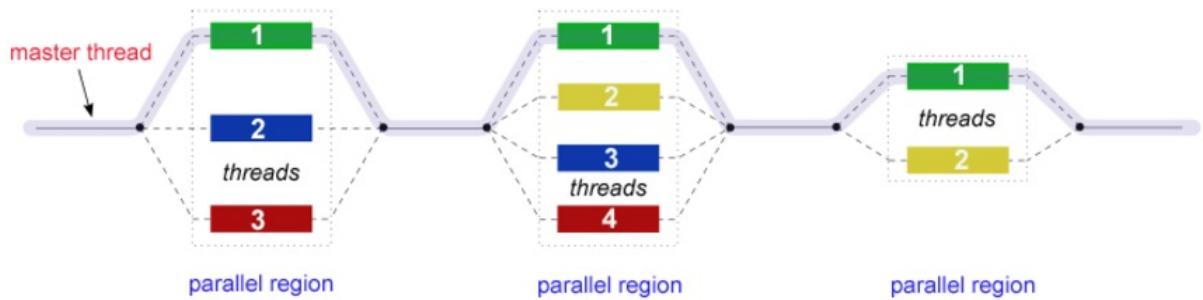
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 void mp_test(void)
6 {
7     int my_rank = omp_get_thread_num();
8     int thread_count = omp_get_num_threads();
9     printf("Hello from thread %d of %d\n", my_rank, thread_count);
10 }
11
12 int main(int argc, char *argv[])
13 {
14     int thread_count = strtol(argv[1], NULL, 10);
15     //parallel block start declaration
16     #pragma omp parallel num_threads(thread_count)
17     {
18         mp_test();
19     }
20     //ends with closing bracket
21     return 0;
22 }
```

Listing 3.3: Prosta aplikacja wykorzystująca dyrektywę **#pragma omp parallel** [28]

Powyższy prosty przykład ilustruje wykorzystanie OpenMP do równoległego uruchomienia `thread_count` wątków, gdzie każdy z nich wykona funkcję `mp_test`. Dodatkowo użyta klauzula `num_threads` modyfikuje dyrektywę tak aby stworzyła tyle wątków ile zostało podane w liście argumentów przy uruchomieniu programu (wskaźnik na tablicę `argv`). Jeśli jeden z wątków wcześniej skończy pracę od innych, czeka aż reszta zakończy wykonywanie funkcji `mp_test`. Każdy ze stworzonych wątków otrzymuje swoje id, stopień oraz parametr określający liczbę innych wątków w ramach tego samego bloku. Korzystając z funkcji `omp_get_thread_num` i `omp_num_threads` (nagłówek `<omp.h>`) otrzymywane jest id i liczba współpracujących wątków. Współdzielonym zasobem pomiędzy wątkami jest strumień wyjściowy `stdout`. [28][2]

Kolekcja wątków wykonujących blok kodu nazywana jest zespołem. Wątek, który przetwarzał in-

strukcje przed dyrektywą `#pragma omp parallel` jest zarządcą(**master**), a dodatkowe wątki są jego podwładnymi(**slave**). Kiedy wszystkie zakończą swoją pracę łączą się z wątkiem twórcą, który wtedy kontynuuje wykonywanie dalszych instrukcji(patrz rys. 3.2).[5][8]



Rysunek 3.2: Model tworzenia wątków w OpenMP[5]

Podobnie jak dla Pthreads OpenMP uwzględnia ochronę sekcji krytycznej oraz metody synchronizacji wątków. Do przeciwdziałania wyścigom i zakleszczeniom(deadlocks) pomiędzy wątkami wykorzystywane są:[8][32]

- Dyrektywa `critical` - tylko jeden wątek może w tym samym czasie wykonywać blok strukturalny. Możliwe jest istnienie wielu sekcji `critical`, gdzie ich nazwy są używane jako globalne identyfikatory. Różne regiony `critical` o tej samej nazwie są traktowane jako jedna sekcja.[28][5]

```
#pragma omp critical [ nazwa ]
{
    blok strukturalny
}
```

- Dyrektywa `atomic` - wykorzystuje specjalne instrukcje sprzętowe, dzięki czemu możliwa jest dużo szybsza realizacja sekcji krytycznej. Atomowe operacje to takie które wykonywane są zawsze całkowicie, bez interwencji innego wątku. Najczęściej sekcje `atomic` używane są do prostych operacji na licznikach zmienianych przez kilka wątków równolegle.[32][8]
Operacje zmiany zmiennej:

```
#pragma omp atomic [ nazwa ]
{
    ++x; --x; x++; x--;
    x += expr; x -= expr; x *= expr; x /= expr; x &= expr;
    x = x+expr; x = x-expr; x = x*expr; x = x/expr; x = x&expr;
    x = expr+x; x = expr-x; x = expr*x; x = expr/x; x = expr&x;
    x |= expr; x ^= expr; x <<= expr; x >>= expr;
    x = x|expr; x = x^expr; x = x<<expr; x = x>>expr;
    x = expr|x; x = expr^x; x = expr<<x; x = expr>>x;
}
```

Operacje czytania i nadpisania zmiennej:

```
#pragma omp atomic [ nazwa ]
{
    var = x++;
}
```

```

    var = x;
    x++;
    x = expr;
}

```

- Blokada `omp_lock_t` z pliku nagłówkowego `<omp.h>` - ogranicza dostęp do funkcji krytycznej. Posiada pięć funkcji do manipulacji blokadą [32][5]:

- `omp_init_lock` - inicjalizuje blokadę. Po tym wywołaniu nie jest jeszcze ustawiona.
- `omp_destroy_lock` - usuwa blokadę, nie może być wcześniej ustawiona.
- `omp_set_lock` - próbuje ustawić blokadę. Jeśli inny wątek już wywołał tę funkcję, czeka aż blokada będzie znowu dostępna, wtedy zostaje ona ustawiona.
- `omp_unset_lock` - zwalnia blokadę, powinna być użyta tylko przez wątek, który ją ustawił. W innym wypadku zachowanie programu będzie niezdefiniowane.
- `omp_test_lock` - próbuje ustawić blokadę. Jeżeli jest już ustawiona przez inny wątek, zwraca 0. Jeśli nie, blokuje sekcję krytyczną i zwraca 1.

OpenMP umożliwia automatyczne podzielenie pomiędzy wątki iteracji pętli `for`. Używana jest do tego dyrektywa `#pragma omp for`, która rozdziela na sekcję rozpatrywaną pętlę, pomiędzy wszystkie stworzone wątki w ramach tego bloku strukturalnego. [32][28]

```

#pragma omp parallel num_threads(n)
{
    #pragma omp for
    {
        for(int i = 0; i < 10; i++)
        {
            cout << i << endl;
        }
    }
}

```

Zasięg zmiennych jest zależny od tego czy są one zdefiniowane przed blokiem strukturalnym, czy wewnątrz niego. Zmienne zdefiniowane przed dyrektywą `parallel` albo `for`, są widoczne dla każdego wątku. Te których deklaracje są wewnątrz konstrukcji OpenMP, są prywatne dla każdego stworzonego wątku w tym bloku. Będą one zawierać indywidualne kopie tej zmiennej we własnej pamięci stosu. [28][8]

Podsumowując, OpenMP jest biblioteką wyższego poziomu, na którą składa się zestaw dyrektyw preprocesora, makr i funkcji umożliwiających tworzenie aplikacji wielowątkowych. Jest dobrą alternatywą dla biblioteki Pthreads, ponieważ podobnie jak ona jest napisana dla języka C i dodatkowo umożliwia korzystanie z nowszych funkcjonalności języka C++. Upraszcza synchronizację i tworzenie nowych wątków oraz umożliwia bez zmiany kodu, wykonanie instrukcji programu sekwencyjnie dla kompilatorów niewspierających dyrektyw `#pragma`. W przeciwieństwie do Pthreads nie wymaga określenia konkretnej funkcji, którą ma wykonywać dany wątek, tylko automatycznie rozdziela wykonywanie instrukcji bloku strukturalnego(fragment kodu objęty dyrektywą `code#pragma`) przez ustawioną liczbę stworzonych wątków. Umożliwia inkrementalne zwiększanie równoległości programu poprzez dodawanie kolejnych dyrektyw i elementów biblioteki zdefiniowanych w nagłówku `<omp.h>`

3.1.3 Wielowątkowość w standardzie C++11

To jest podrozdział 3 rozdziału 1. Nowy standard języka C++ - C++11, istotnie zmienił podejście do pisania programów w porównaniu do starszej wersji C++98. Zamiarem komisji standaryzacyjnej było stworzenie bardziej czytelnego, prostszego w pisaniu oraz bardziej zoptymalizowanego języka. Wśród wielu nowości, takich jak inteligentne wskaźniki, operator przenoszenia, konstruktor przenoszący, wyrażenia lambda, dedukcja typu auto; jednym z najistotniejszych było wprowadzenie współbieżności do biblioteki standardowej. Rezultatem tego jest umożliwienie tworzenia wieloplatformowych aplikacji wielowątkowych bez potrzeby używania dodatkowych bibliotek, takich jak Pthreads, Windows threads, OpenMP.[25][30]

Biblioteka standardowa C++11 umożliwia dwie metody wykonywania zadań asynchronicznie[25][15]:

- Z wykorzystaniem obiektów typu `std::thread`
- używając podejścia zadaniowego dla obiektów `std::future`

Podobnie jak dla Pthreads, wielowątkowość implementowana z pomocą `std::thread`, zakłada tworzenie nowych wątków z unikalnymi id, pamięcią stosu, funkcją wykonawczą oraz listą argumentów do tej funkcji. Każdy stworzony wątek może być typu `joinable`, albo `detached`. `Joinable` oznacza, że wątek powinien zakończyć działanie przed wywołaniem swojego destruktora. Niezbędne jest zastosowanie do tego przez wątek twórcy metody `join()`, która zapewnia, że stworzony wątek wykona swoje zadanie zanim zostanie zniszczony. Jeśli pożądane jest pozwolenie wątkowi na pracę po wywołaniu jego destruktoru, używana jest metoda `detach()`. [15][30]

Analogicznie do POSIX threads, do synchronizacji wykorzystywane są blokady w postaci muteksów `std::mutex` oraz zmienne warunkowe do generacji zdarzeń `std::condition_variable`. W celu usunięcia konieczności ręcznego ustawiania i odblokowywania mutexa (`mutex.lock()`, `mutex.unlock()`), wykorzystywany jest obiekt `std::lock_guard`, który po inicjalizacji muteksem jako argument, ustawia blokadę i zapewnia jej usunięcie po wyjściu z zasięgu swojej deklaracji (patrz listing 3.4). [15][16]

```
1 #include <thread>
2 #include <mutex>
3 #include <iostream>
4 #include <utility>
5 #include <chrono>
6 #include <functional>
7 #include <atomic>
8
9 using namespace std;
10
11 int protected_global = 0;
12 std::mutex protected_global_mutex;
13
14 void test_fun(int n)
15 {
16     for (int i = 0; i < n; ++i) {
17         std::cout << "Thread 1 executing\n";
18         std::this_thread::sleep_for(std::chrono::milliseconds(10));
19     }
20 }
21
22 void safe_increment(int a, std::string &str)
23 {
24     std::lock_guard<std::mutex> lock(protected_global_mutex);
25     protected_global += a;
26
27     cout << "Thread " << std::this_thread::get_id() << " incremented protected_global by
    " << a << '\n';
```

```

28     cout << str << endl;
29     // mutex is unlocked after lock_guard leaves current scope
30 }
31
32 int main()
33 {
34     std::cout << "Init global : " << protected_global << '\n';
35     std::thread t1; //declaration, new thread was not created
36     //after giving execute function new thread starts running
37     std::thread t2(test_fun, 2);
38     std::thread t3(std::move(t2)); //calls move constructor
39     //t2 is not a thread anymore, t3 continues executing test_fun
40
41     //passing reference arguments, testing lock_guard use
42     std::string s1 = "Kamehameha!!";
43     std::string s2 = "Hadoken!!";
44     std::thread t4(safe_increment, 5, std::ref(s1));
45     std::thread t5(safe_increment, 10, std::ref(s2));
46
47     //joinin with main thread
48     t3.join();
49     t4.join();
50     t5.join();
51
52     std::cout << "Final global : " << protected_global << '\n';
53 }

```

Listing 3.4: Podstawowe funkcjonalności `std::thread` [13]

Innym podejściem do współbieżności, które zostało zawarte w nowym standardzie, jest współbieżność zadaniowa. Daje ona możliwość wykonania zadania - funkcji i zwraca jeden wynik. Funkcja `std::async()` odpowiada za asynchroniczne uruchomienie podanej w liście argumentów funkcji. Zwraca obiekt typu `std::future`, z którego możemy otrzymać wynik używając metody `get`. Zaletą tego podejścia jest jego prostota, kiedy nie potrzebujemy skomplikowanej synchronizacji pomiędzy wątkami, tylko wiemy, że każdy z nich ma wykonać równolegle niezależne zadanie(patrz listing 3.5). [25][30]

```

1 #include <iostream>
2 #include <future>
3 #include <thread>
4
5 int main()
6 {
7     // future from a packaged_task
8     std::packaged_task<int> task([]() { return 7; }); // wrap the function
9     std::future<int> f1 = task.get_future(); // get a future
10    std::thread(std::move(task)).detach(); // launch on a thread
11
12    // future from an async()
13    std::future<int> f2 = std::async(std::launch::async, []() { return 8; });
14
15    // future from a promise
16    std::promise<int> p;
17    std::future<int> f3 = p.get_future();
18    std::thread([ &p]{ p.set_value_at_thread_exit(9); }).detach();
19
20    std::cout << "Waiting..." << std::flush;
21    f1.wait();
22    f2.wait();
23    f3.wait();

```



```
24 |     std::cout << "Done!\nResults are: "  
25 |         << f1.get() << ' ' << f2.get() << ' ' << f3.get() << '\n';  
26 | }
```

Listing 3.5: Przykład zastosowania `std::future` i `std::async()` [13]

3.2 Programowanie równoległe z wykorzystaniem GPU

To jest rozdział 2

3.2.1 Architektura GPU i porównanie względem CPU

To jest podrozdział 1 rozdziału 2

3.2.2 Biblioteka OpenCL

To jest podrozdział 2 rozdziału 2

Rozdział 4

Algorytm Viterbiego

4.1 Opis działania i zastosowania

To jest rozdział 1

4.2 Implementacja w języku C++

To jest rozdział 2

4.2.1 Wersja szeregową

To jest podrozdział 1 rozdziału 2

4.2.2 Wersja równoległa - C++11

To jest podrozdział 2 rozdziału 2

4.2.3 Wersja równoległa - OpenCL

To jest podrozdział 3 rozdziału 2

Rozdział 5

Wyniki badań doświadczalnych implementacji algorytmu Viterbiego

5.1 Porównanie czasu działania dla implementacji szeregowej, wielowątkowej oraz z wykorzystaniem biblioteki OpenCL

To jest rozdział 1

5.2 Porównanie szybkości algorytmów dla różnych konfiguracji sprzętowych

To jest rozdział 2

Rozdział 6

Wnioski końcowe

Rozdział 7

Załącznik B

To jest załącznik B

Rozdział 8

Załącznik A

To jest załącznik A

Spis rysunków

2.1	Przykład zautomatyzowanej linii technologicznej wykorzystującej system wizyjny[31] . . .	4
2.2	Przykład obrazów używanych w testowaniu pozycji i orientacji elementów[12]	5
2.3	Przykład wizyjnej identyfikacji[12]	5
2.4	Procesor <i>QorlQ Layerscape LS1028</i> do aplikacji przemysłowych firmy NXP, wyposażony w dwa rdzenie ARMv8[27]	6
2.5	Struktura mikrokontrolera <i>STM32F401CC</i> [29]	7
3.1	Wykorzystanie <code>pthread_join</code> do synchronizacji wątków[6]	9
3.2	Model tworzenia wątków w OpenMP[5]	13

Listings

3.1	Przykład tworzenia i uruchamiania wątków	9
3.2	Przykład wykorzystanie muteksa do synchronizacji aplikacji wielowątkowej	10
3.3	Prosta aplikacja wykorzystująca dyrektywę <code>#pragma omp parallel</code> [28]	12
3.4	Podstawowe funkcjonalności <code>std::thread</code> [13]	15
3.5	Przykład zastosowania <code>std::future</code> i <code>std::async()</code> [13]	16

Rozdział 9

Bibliografia

- [1] Sachin B. Bhosale Amol N. Dumbare, Kiran P.Somase. Mobile robot for object detection using image processing. *International Journal of Advane Research in Computer Science and Managment Studies*, 1(6):81–84, 2013.
- [2] Dieter an Mey. Parallel programming in openmp introduction. <http://scc.ustc.edu.cn/zlsc/cxyy/200910/W020100308601022991415.pdf>.
- [3] Atmel. Atmel avr 8-bit and 32-bit microcontrollers. <http://www.atmel.com/products/microcontrollers/avr/default.aspx>.
- [4] Blaise Barney. Introduction to parallel computing. https://computing.llnl.gov/tutorials/parallel_comp/.
- [5] Blaise Barney. Openmp. <https://computing.llnl.gov/tutorials/openMP>.
- [6] Blaise Barney. Posix threads programming. <https://computing.llnl.gov/tutorials/pthreads/>.
- [7] Basler. Basler camera portfolio. <https://www.baslerweb.com/en/products/cameras/>.
- [8] OpenMP Architecture Review Board. Openmp application program interface. <http://www.openmp.org/wp-content/uploads/spec25.pdf>, 2005.
- [9] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997. ISBN:0201633922.
- [10] Pong P. Chu. *FPGA Prototyping by VHDL examples*. John Wiley and Sons, Inc., 2008. ISBN:9780470185315.
- [11] Cognex. Insight 5000 industrial vision systems. <http://www.cognex.com/productstemplate.aspx?id=13915>.
- [12] Cognex. Introduction to machine vision. http://www.assemblymag.com/ext/resources/White_Papers/Sep16/Introduction-to-Machine-Vision.pdf, 2016.
- [13] C++ Concurrency. C++ reference documentation. <http://en.cppreference.com/>.
- [14] E.R. Davies. *Computer and Machine Vision: Theory, Algorithms, Practicalities*. Elsevier, 225 Wyman Street, Waltham, 02451, USA, 2012. ISBN:9780123869081.
- [15] Standard C++ Foundation. C++11 standard library extensions - concurrency. <https://isocpp.org/wiki/faq/cpp11-library-concurrency>.
- [16] Standard C++ Foundation. Multi-threaded programming: C++11. http://www.bogotobogo.com/cplusplus/multithreaded4_cplusplus11.php.

- [17] Ian Grout. *Digital Systems Design with FPGAs and CPLDs*. Elsevier, 2008. ISBN:9780750683975.
- [18] Przemysław Mazurek Grzegorz Matczak. Line following with real-time viterbi trac-before-detect algorithm. *Przegląd Elektrotechniczny*, 1/2017:69–72, 2017.
- [19] National Instruments. Choosing the right camera bus. *NI white papers*, 2016.
- [20] Intel. Intel processors and chipsets for embedded applications. <http://www.intel.pl/content/www/pl/pl/intelligent-systems/embedded-processors-which-intel-processor-fits-your-project.html>.
- [21] David Patterson John Hennessy. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011. ISBN:9780123838728.
- [22] Mike Houston Katvon Fatahalian. A closer look at gpus. *Communications of the ACM*, 51(10):50–57, 2008.
- [23] Guy Kerens. Multi-threaded programming with posix threads. <http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html>.
- [24] K.N. King. *Język C. Nowoczesne programowanie*. Helion, 2008. ISBN:9788324628056.
- [25] Scott Meyers. *Effective Modern C++*. O'Reilly, 2015. ISBN:9781491903995.
- [26] Nvidia. What is gpu-accelerated computing. <http://www.nvidia.com/object/what-is-gpu-computing.html>.
- [27] NXP. Arm technology-based solutions - nxp microcontrollers and processors. <http://www.nxp.com/products/microcontrollers-and-processors/arm-processors:ARM-ARCHITECTURE>.
- [28] Peter S. Pacheco. *An Introduction to Parallel Programming*. Elsevier, 2011. ISBN:9780123742605.
- [29] ST. Stm32 32-bit arm cortex mcus. <http://www.st.com/en/microcontrollers/stm32-32-bit-arm-cortex-mcus.html?querycriteria=productId=SC1169>.
- [30] Bjarne Stroustrup. *Język C++.Kompendium wiedzy*. Helion, 2013. ISBN:9788324685301.
- [31] Andy Wilson. Industrial inspection : Line-scan-based vision system tackles color print inspection. *Vision Systems Design*, 2014.
- [32] Joel Yliluoma. Guide into openmp: Easy multithreading programing for c++. <http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html>.