

**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

<b>1</b>	<b>Streszczenie</b>	<b>3</b>
<b>2</b>	<b>Wstęp</b>	<b>4</b>
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
<b>3</b>	<b>Metody równoległego przetwarzania danych</b>	<b>8</b>
3.1	Współbieżność w ramach architektury CPU . . . . .	8
3.2	Wielowątkowość aplikacji dla języka C/C++ . . . . .	13
3.2.1	Biblioteka POSIX dla systemów Unix . . . . .	13
3.2.2	OpenMP - wieloplatformowe API . . . . .	16
3.2.3	Wielowątkowość w standardzie C++11 . . . . .	19
3.3	Programowanie równoległe z wykorzystaniem GPU . . . . .	22
3.3.1	Architektura GPU . . . . .	22
3.3.2	Biblioteka OpenCL . . . . .	23
<b>4</b>	<b>Algorytm Viterbiego</b>	<b>28</b>
4.1	Opis działania i zastosowania . . . . .	28
4.2	Wykrywanie linii na obrazie cyfrowym z wykorzystaniem algorytmu Viterbiego . . . . .	31
4.3	Implementacja detekcji linii języku C++ . . . . .	32
4.3.1	Wersja szeregową . . . . .	32
4.3.2	Wersja równoległa - C++11 . . . . .	34
4.3.3	Wersja równoległa - OpenCL . . . . .	35
<b>5</b>	<b>Wyniki badań doświadczalnych implementacji algorytmu Viterbiego</b>	<b>36</b>
5.1	Porównanie czasu działania dla implementacji szeregowej, wielowątkowej oraz z wykorzystaniem biblioteki OpenCL . . . . .	36
5.2	Porównanie szybkości algorytmów dla różnych konfiguracji sprzętowych . . . . .	36
<b>6</b>	<b>Wnioski końcowe</b>	<b>37</b>
<b>7</b>	<b>Załącznik B</b>	<b>38</b>
<b>8</b>	<b>Załącznik A</b>	<b>39</b>
	Spis rysunków	39
<b>9</b>	<b>Bibliografia</b>	<b>41</b>

# Rozdział 1

## Streszczenie

To jest streszczenie

# Rozdział 2

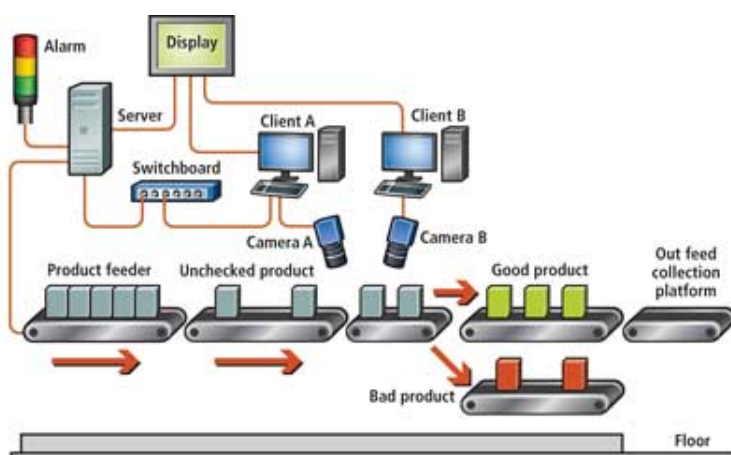
## Wstęp

### 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 [13]:

- 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. [13][15]



Rysunek 2.1: Przykład zautomatyzowanej linii technologicznej wykorzystującej system wizyjny[48]

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.[13]



Rysunek 2.2: Przykład obrazów używanych w testowaniu pozycji i orientacji elementów[13]

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.[13]



Rysunek 2.3: Przykład wizyjnej identyfikacji[13]

## 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 [23]. Na podstawie tego można stwierdzić, że główny problem będzie stanowił czas obliczenia sterowania i od niego będzie zależeć szybkość działania systemu wizyjnego[12][7].

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

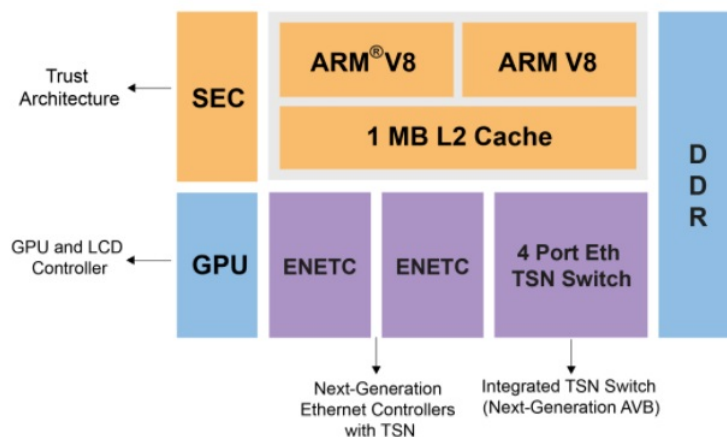
- Sprzężenie niezbędne do obliczenia zmiany położenia i prędkości robota.
- Lokalizację przeszkód oraz innych robotów (Swarm Robotics).
- Analizę oraz monitorowanie otoczenia.

[19][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 [34] [24].

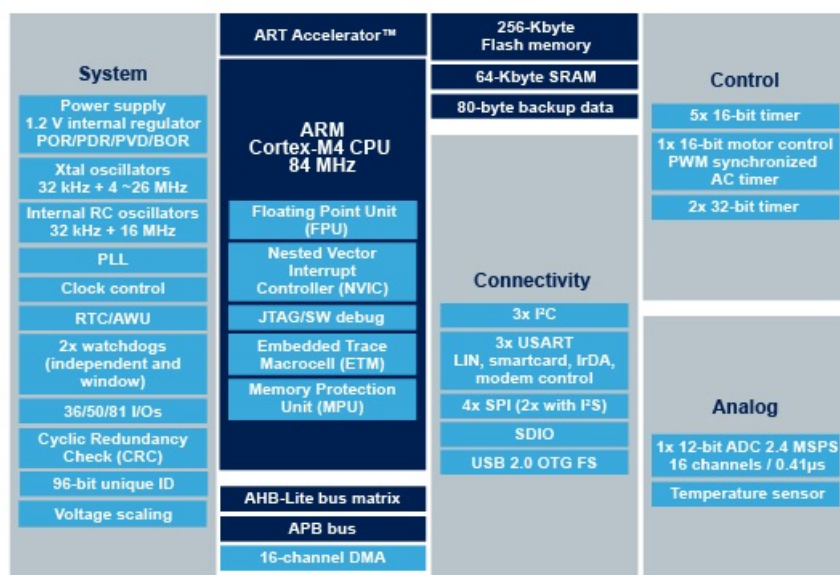
**QorIQ® Layerscape LS1028 A Block Diagram**



Rysunek 2.4: Procesor *QorIQ Layerscape LS1028* do aplikacji przemysłowych firmy NXP, wyposażony w dwa rdzenie ARMv8[34]

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. [18][11].

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) [43] [3].



Rysunek 2.5: Structura mikrokontrolera *STM32F401CC*[43]

Zewnętrzna jednostka obliczeniowa pozwala na wykorzystanie możliwości konwencjonalnych wielordzeniowych procesorów dla komputerów PC oraz procesora karty graficznej, który jest wyspecjalizowany w obliczeniach równoległych[33] [25][26]. 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

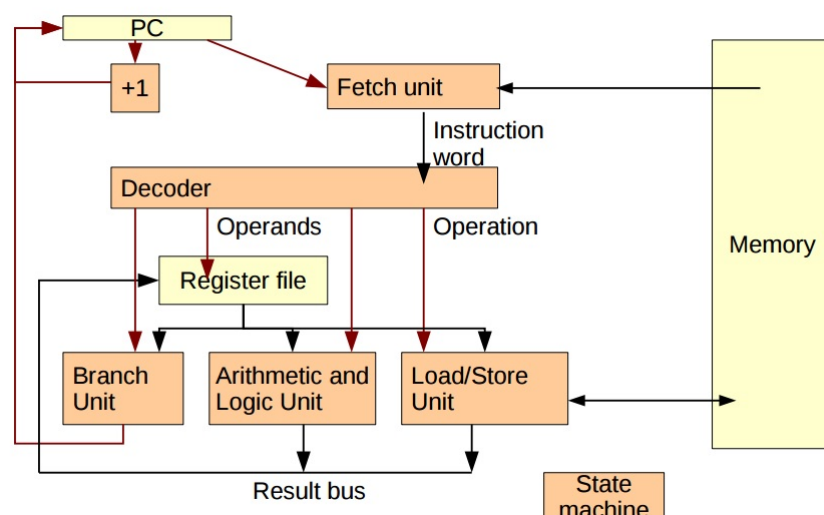
# Metody równoległego przetwarzania danych

### 3.1 Współbieżność w ramach architektury CPU

Rozwój technologii wytwarzania układów elektronicznych, doprowadził do rozpowszechnienia mikroprocesorów, które zawierają wszystkie komponenty w jednym układzie scalonym. Ponadto pojedynczy układ zawiera obecnie więcej niż jeden procesor - rdzeń, każdy z nich posiada 2 wątki sprzętowe. Powoduje to coraz większą popularność wykorzystywania współbieżnego modelu programowania współczesnych wielordzeniowych procesorów.[44] [25]

Każdy procesor składa się z :

- jednostki arytmetyczno logicznej - ALU(arithmetic logical unit).
- zestawu rejestrów
- jednostki kontrolnej - CU(control unit)



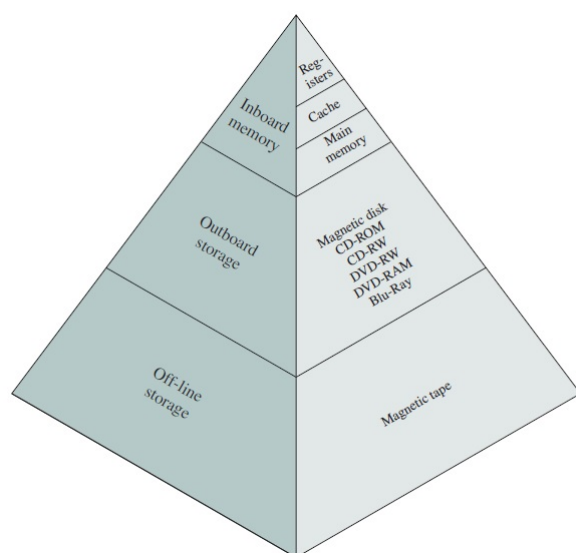
Rysunek 3.1: Schemat cyklu pobierania i wykonania instrukcji przez CPU [44]

Jednostka arytmetyczno logiczna(**ALU**) jest odpowiedzialna za wykonywanie obliczeń. Składa się z jednostek całkowitoliczbowej **IU** i zmiennoprzecinkowej(**FPU**), które wykonują operacje arytmetyczne



i logiczne na bazie otrzymanych instrukcji z pamięci komputera. Adres do instrukcji jest przechowywany w specjalnym rejestrze - zwany licznikiem programu **PC**. Po pobraniu instrukcji z pamięci jest ona następnie przekazywana do rejestru instrukcji **IR**, po czym zwiększany jest licznik programu, tak aby wskazywał na następną instrukcję. Dane z rejestru instrukcji są później dekodowane, tak aby określić jaką operację przeprowadzić. Następnie jednostka kontrolna procesora (**CU**) przekazuje informacje **ALU** jeśli instrukcja dotyczyła operacji arytmetycznych. W innym wypadku jeśli procesor miał dokonać operacji na pamięci (np. załadować wartość do rejestru ogólnego przeznaczenia **GPR**) zostaje ona wtedy wykonana wykorzystując jednostki zapisu/odczytu pamięci (**load/store unit**). [44][45]

Szybkość wykonywania operacji jest w obecnych procesorach bezpośrednio powiązana z szybkością odczytu i zapisu danych pamięci komputera. Głównym problemem doboru rodzaju pamięci do zastosowania jest jej cena w zależności od pojemności i szybkości dostępu. Im większa pojemność tym wolniejsza i tańsza pamięć. Optymalizacja kosztów, szybkości oraz ilości pamięci została przeprowadzona poprzez zastosowanie hierarchicznego modelu pamięci. Niższy poziom oznacza mniejszy koszt, zwiększenie pojemności, zwiększenie czasu dostępu, zmniejszenie częstotliwości z jaką procesor będzie wykorzystywał tą pamięć. [25][44] Dlatego mniejsza, droższa i szybsza pamięć jest uzupełniana przez większe i tańsze.

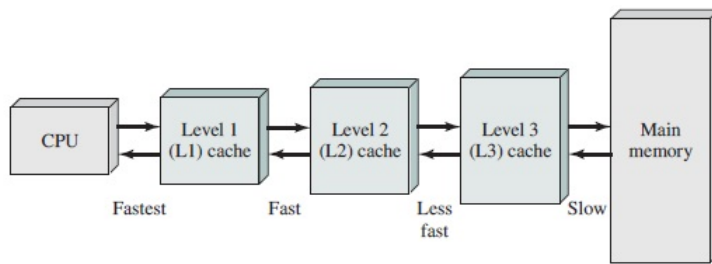


Rysunek 3.2: Hierarchia pamięci [44]

Na samym szczycie hierarchii stoją rejestry procesora, w których zawierają się instrukcje, dane i adresy z pamięci pobierane z pamięci niższego poziomu, wykorzystywane przez **ALU** oraz jednostkę zapisu i odczytu. Jako bufor do wymiany danych z pamięcią główną, jest wykorzystywana pamięć podręczna procesora - CPU cache (patrz rys. 3.3). Cache jest obecnie najczęściej podzielony na trzy poziomy - L1, L2, L3. Każdy z nich posiada kopię fragmentu pamięci głównej, gdzie największy jest zawarty w L3 cache, a najmniejszy w L1. Jeśli procesor potrzebuje pobrać dane z adresu w pamięci głównej, którego kopia nie jest w L1, sprawdzany jest cache L2, następnie L3.

Poprzez zastosowanie hierarchicznego modelu pamięci, zwiększyła się istotność w programowaniu sekwencyjnym jak i współbieżnym, brania pod uwagę budowy pamięci podręcznej. Cache składa się z linii odpowiadających blokom z pamięci głównej. Jeśli potrzebny adres bloku pamięci głównej nie występuje w cache'u, musi być pobrany z pamięci głównej, zapisany w nim oraz przekazany do rejestrów procesora. Dlatego, że cache składa się z kopii kolejnych bloków pamięci głównej, przy operacji na tablicach w językach programowania w celu wykorzystania szybkości pamięci podręcznej, należy je przetwarzać wierszami, a nie kolumnami; zgodnie z kolejności występowania w pamięci. [25][44][45]

Podstawową metodą zwiększania szybkości wykonywania instrukcji przez procesor jest wykorzysty-



Rysunek 3.3: Trójpoziomowa organizacja pamięci cache [44]

wanie potokowości. Zgodnie z wcześniej przedstawionym modelem wykonywania instrukcji, każda jest pobierana z pamięci, dekodowana, wykonywana przez procesor, a wynik jest zapisywany do określonego rejestru. Procesory jedno-cyklowe, które wykonują wszystkie powyższe kroki w czasie jednego cyklu zegara są proste w budowie, ale zużywają dużo zasobów sprzętowych ponieważ procesor nie przetwarza więcej niż jednej instrukcji w tym samym czasie.[45]. Stoując mechanizm potokowości(pipelining) cykl przetwarzania instrukcji jest podzielony na odrębne fazy:

1. Fetch - pobranie instrukcji z pamięci
2. Decode - interpretacja instrukcji
3. Execute - wykonanie instrukcji
4. Write - zapisanie wyniku do rejestru

Gdy pierwsza zostaje zakończona, zaczyna się druga, a pierwsza zaczyna pobierać następną instrukcję. Analogicznie po dekodowaniu pierwszej instrukcji jest ona wykonywana w 3-ciej fazie, a w tym samym czasie druga instrukcja zaczyna być dekodowana. Zgodnie z tym tokiem postępowania w czasie pierwszego cyklu zegara procesora pobrane zostaną 4 instrukcje, 3 z nich zostaną zinterpretowane, 2 wykonane i wynik jednej będzie zapisany w rejestrze. Dla następnego cyklu poprzednio nie przetworzone instrukcje zostaną

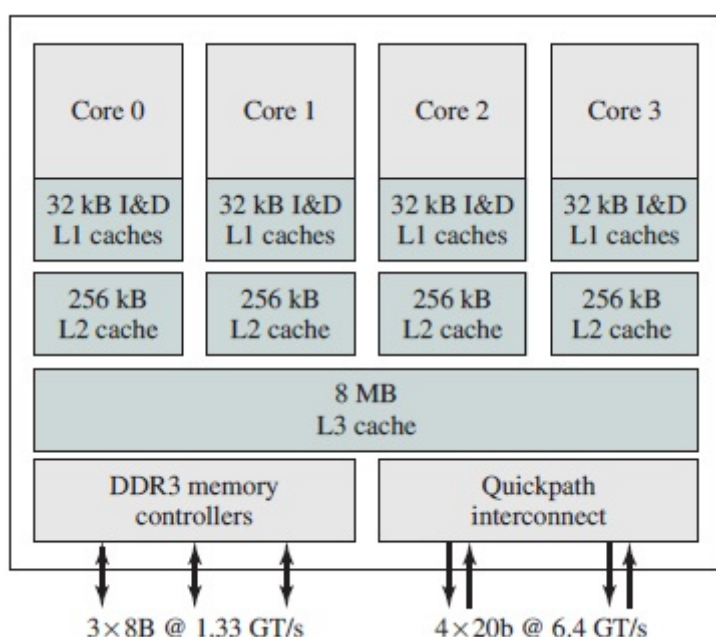


Rysunek 3.4: Przykład 4-fazowego potoku(pipeline) procesora [45]

dokończone oraz kolejne 3 zostaną częściowo przetworzone, a pierwsza z nowych instrukcji przejdzie przez wszystkie 4 fazy (patrz rys.3.4). [45][25]

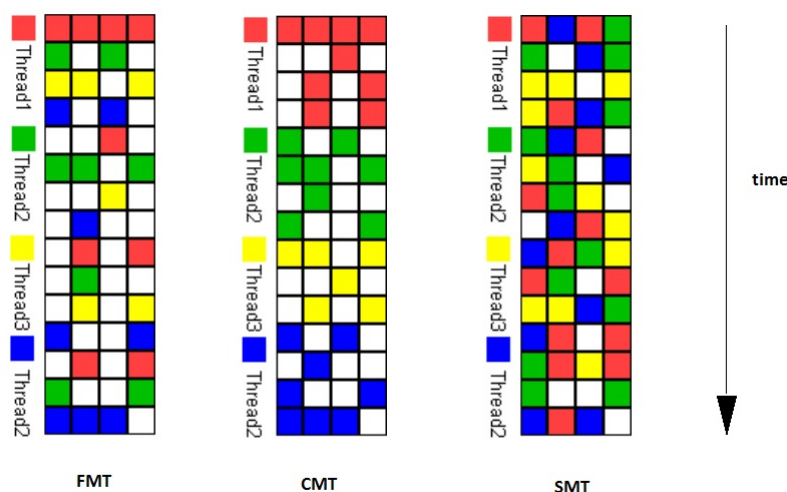
Wadą zastosowanie potokowości jest zwiększenie złożoności logiki sterowania procesora, ze względu na synchronizację faz przetwarzania instrukcji. Dodatkowo często może dojść do zablokowania jednej z faz potoku, co powoduje opóźnienie w wykonywaniu kolejnych instrukcji. Ponadto szybkość potoku jest zależna od najwolniejszej z faz, która staje się wąskim gardłem całego procesu. Powoduje to sztuczne wydłużenie czasu wykonania pozostałych faz, co bezpośrednio oznacza marnowanie zasobów sprzętowych procesora. Właśnie dlatego, aby koszt zastosowania potokowości odpowiadał wzrostowi wydajności procesora, wymagane jest od projektanta CPU zbalansowanie czasu poszczególnych faz potoku. Z tego powodu z czasem zaczęto odchodzić od zwiększania osiągnięć procesora z wykorzystaniem potokowości. Dalsza współbieżność wykonywania instrukcji programu została oparta na wykorzystaniu technologii wielordzeniowych.[45][25]

Innym podejściem do paralelizmu jest sprzętowa implementacja wielowątkowości - **TLP**(Thread-level parallelism). Występują trzy sprzętowe podejścia do realizacji wielowątkowości[25] (patrz rys.3.6):



Rysunek 3.5: Intel Core i7, przykład wielordzeniowego procesora wykorzystującego model **SMT** [44]

- Fine-grained multithreading(FMT) - przełączanie pomiędzy wątkami występuje co każdy cykl procesora. Powoduje to naprzemienne wykonywanie instrukcji, gdzie w przypadku wątków, które czekają na zdarzenie i nie wykonują instrukcji, są one w tym czasie pomijane. Zaletą tego podejścia jest zmniejszenie straty wydajności, spowodowanych przez czekające wątki. Wadą jest zwolnienie wykonywania instrukcji pojedynczego wątku.
- Coarse-grained multithreading(CMP) - alternatywa dla FMT, zmienia obecnie wykonywany wątek, tylko w momencie długotrwałego oczekiwania na kontynuowanie egzekucji(np. brak potrzebnego adresu w cache L2 lub L3). Zaletą tego podejścia jest zmniejszenie czasu wykonania pojedynczego wątku.
- Simultaneous multithreading(SMT) - najpopularniejsza implementacja wielowątkowości dla CPU, ulepszona wersja FMT dla procesorów wielordzeniowych o dynamicznym przydzielaniu. Pozwala ono na wykonywanie w tym samym cyklu instrukcji z kilku wątków. Pozwala to na optymalne wykorzystanie możliwości procesora do współbieżnego przetwarzania instrukcji.



Rysunek 3.6: Rodzaje implementacji **TLP**

Oprócz omówionych wcześniej metod implementacji współbieżności: potokowość - równoległość na poziomie instrukcji (**ILP** - instruction-level parallelism), **TLP**, występuje trzecia kategoria - współbieżność na poziomie danych (**DLP** - Data-level parallelism). Polega on na równoczesnym przetwarzaniu wielu strumieni danych. Wyróżniane są architektury:

- MIMD - *multiple instruction, multiple data*
- SIMD - *single instruction multiple data*

Systemy MIMD wspierają jednocześnie wykonywanie wielu instrukcji, operując na wielu strumieniach danych. Składają się z kolekcji niezależnych procesorów lub rdzeni, gdzie każdy posiada własny zestaw rejestrów, ALU i jednostkę kontrolną. Systemy te nie posiadają jednego zegara synchronizującego wszystkie procesory, każdy z nich może pracować asynchronicznie.[37][44]. Występują dwa rodzaje systemów MIMD: współdzielące pamięć(*shared-memory systems*) oraz z oddzielną pamięcią(*distributed-memory systems*). Pierwsze z nich są kolekcją autonomicznych procesorów połączonych szyną pamięci głównej. Komunikują się ze sobą najczęściej używając współdzielonych obszarów pamięci. W systemach z oddzielną pamięcią, każdy procesor posiada własną pamięć prywatną, a komunikacja jest wykonywana poprzez wykorzystywanie specjalnych funkcji sygnalizujących.[44]

Architektura SIMD pozwala wykorzystywać jedną instrukcję do przetwarzania wielu danych. Może być on rozpatrywany jako jedna jednostka sterująca wyposażona w wiele jednostek ALU. Posiada najczęstsze zastosowanie w dokonywaniu operacji na tablicach oraz współbieżnego przetwarzania pętli. [25][37]

## 3.2 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.[25][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 serty, 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][27] Wtedy ze względu na problem błędnego odczytu, bądź zapisu wartości w pamięci kiedy inny wątek ją już napisał, 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. [10][46]

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.[10][46]

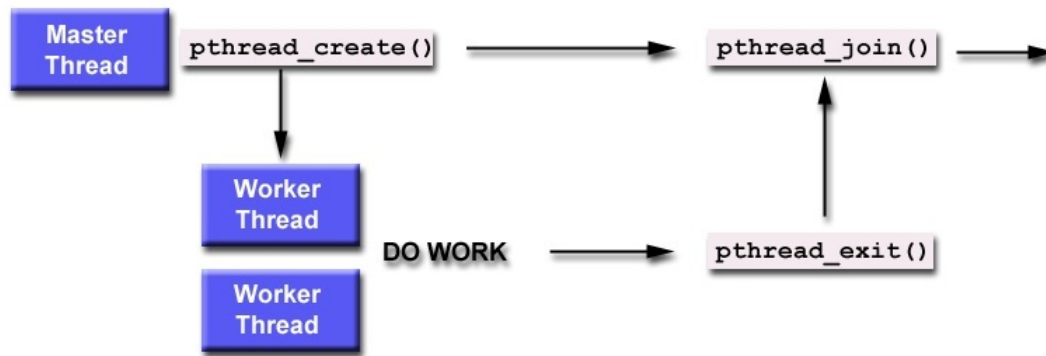
W tym rozdziale zostaną omówione różne metody tworzenia aplikacji wielowątkowych w języku C/C++. Skupiono się na bibliotekach dla tych języków programowania ze względu na ich popularność w tworzeniu rozwiązań dla systemów wbudowanych, która wynika z wysokiej wydajności kodu i małego zużycia pamięci w porównaniu do języków interpretowanych, np. Java, Python.[46][28]

### 3.2.1 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.[10][6][27] 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`. [10]

Jedną z podstawowych metod synchronizacji pomiędzy wątkami jest użycie funkcji `pthread_join`,



Rysunek 3.7: Wykorzystanie `pthread_join` do synchronizacji wątków[6]

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).

Listing 3.1: Przykład tworzenia i uruchamiania wątków

```

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 }

```

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 mutes (mutual exclusion). Mutes jest szczególnym przypadkiem semaforu Dijkstry - semaforem binarnym o zbiorze wartości 0, 1.[10][27][6] W bibliotece POSIX threads mutes jest reprezentowany jako zmienna typu `pthread_mutex_t`. W celu posiadania globalnego zasięgu deklarowana jest jako zmienna `static` lub `extern`. [10][28](patrz 3.2) W celu deklaracji mutesa wykorzystywane jest makro `PTHREAD_MUTEX_INITIALIZER`(patrz . Jeśli mutes 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 mutesa, zwolnienie zaalokowanej pamięci, poprzez wykorzystanie funkcji `pthread_mutex_destroy`. [10]

Listing 3.2: Przykład wykorzystanie mutesa do synchronizacji aplikacji wielowątkowej

```

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     {
45         fprintf(stderr, "Error - pthread_create() return code: %d\n", iret2);
46         return iret2;
47     }
48
49     printf("pthread_create() for thread 1 returns: %d\n", iret1);
50     printf("pthread_create() for thread 2 returns: %d\n", iret2);
51     //wait for threads to finish
52     pthread_join( thread1, NULL);
53     pthread_join( thread2, NULL);
54
55     return 0;
56 }

```

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.[27]

### 3.2.2 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 równoległe. [37][9] 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 równoległe. 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 równoległe zostaną obsługane przez jeden wątek. [37][28][9][49] 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).[37][49](patrz 3.3)



Listing 3.3: Prosta aplikacja wykorzystująca dyrektywę `#pragma omp parallel` [37]

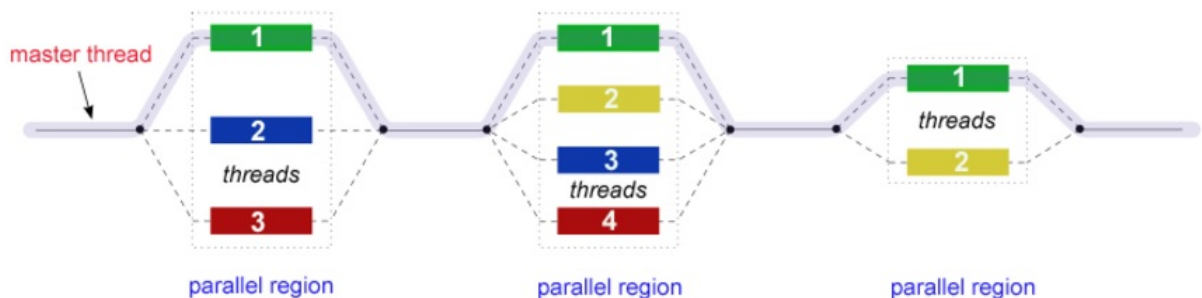
```

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 }

```

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`. [37][2]

Kolekcja wątków wykonujących blok kodu nazywana jest zespołem. Wątek, który przetwarzał instrukcje 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. 3.8). [5][9]



Rysunek 3.8: 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: [9][49]

- 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.[37][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 równoległe.[49][9]

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 [49][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 tą 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. [49][37]

```

#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. [37][9]

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>`

### 3.2.3 Wielowątkowość w standardzie C++11

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.[30][46]

Biblioteka standardowa C++11 umożliwia dwie metody wykonywania zadań asynchronicznie[30][16]:

- 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 destruktor, używana jest metoda `detach()`. [16][46]

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).[16][21]

Listing 3.4: Podstawowe funkcjonalności `std::thread` [14]

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

Innym podejściem do współbieżności, które zostało zawarte w nowym standardzie, jest współbieżność zadaniowa. Daje ona możliwość wykonania zadania - funkcji i następnie zwraca jeden wynik. Wsparcie do tego modelu jest zaimplementowane w postaci [14][30] (patrz listing 3.5) :

- Typów `std::future` i `std::promise`. Pierwszy z nich zawierać będzie wynik zadania. Po zakończeniu zadania drugi jest używany do odczytywania tej wartości.
- `std::packaged_task<T>` - pakuje obiekt typu T do wykonania jako zadanie. Jego konstruktor przyjmuje przy inicjalizacji funkcję, która ma zostać wykonana asynchronicznie. Wynik otrzymywany ze stworzonego na bazie funkcji `get_future()`, obiektu `std::future` używając metody `get()`.
- funkcji `std::async()` - odpowiada za asynchroniczne uruchomienie funkcji podanej w liście argumentów funkcji. Zwraca obiekt typu `std::future`, z którego możemy otrzymać wynik używając metody `get`

Listing 3.5: Przykład zastosowania `std::future` i `std::async()` [14]

```
1  #include <iostream>
2  #include <thread>
3  #include <future>
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     task(); // 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]{ return p.set_value(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 }
```

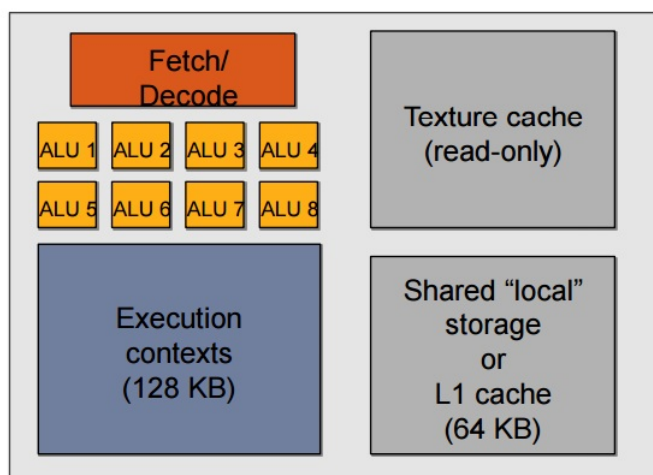
Zaletą tego podejścia jest jego prostota. Kiedy nie potrzebujemy skomplikowanej synchronizacji pomiędzy wątkami, tylko wiemy, że każdy z nich ma wykonać równolegle niezależne zadanie, jest to wygodniejsza metoda pisania aplikacji wielowątkowych. [30][46]

Reasumując, standard C++11 umożliwił programistom tworzenie aplikacji wielowątkowych z użyciem wyłącznie biblioteki standardowej. Wątki w C++11 wspierają nowe elementy standardu takie jak, funkcje lambda, referencje do rvalue, `std::bind` oraz wiele innych. Jest to dodatkowe ułatwienie pisania programów, gdzie nie musimy przejmować się kompatybilnością wykorzystywanej biblioteki. Wystarczające jest używanie nowego kompilatora wspierającego C++11.

### 3.3 Programowanie równoległe z wykorzystaniem GPU

#### 3.3.1 Architektura GPU

GPU(*graphics processing unit*) jest to procesor wyspecjalizowany pod kątem wykonywania elementów potoku graficznego oraz operacji zmiennoprzecinkowych. Obecne procesory graficzne są procesorami wielordzeniowymi zaprojektowanymi specjalnie do wykonywania równoległych obliczeń. Charakterystyczną metodą osiągania współbieżności jest stosowanie architektury **SIMD** do wykonywania jednej instrukcji dla wielu elementów danych. Dzięki temu możliwe jest dzielenie wykonywania obliczeń poprzez liczne jednostki arytmetyczno-logiczne dla każdego z rdzeni GPU. (patrz rys.3.9)[8][25] Oprócz wielu ALU pojedynczy rdzeń posiada pamięć lokalną oraz pamięć dla stałych. Nie wykorzystywana jest hierarchia pamięci jak dla CPU, gdzie aby przyspieszyć czytanie danych z pamięci głównej stosowane są 3 poziomy pamięci podręcznej cache. GPU posiada dużo szybszą pamięć globalną, wspólną dla wszystkich rdzeni.



Rysunek 3.9: Schemat rdzenia procesora wykorzystującego architekturę SIMD [41]

Stosowanie GPU daje najlepsze rezultaty dla problemów umożliwiającym równoległe przetwarzanie, z dużą ilością operacji arytmetycznych w porównaniu do operacji wymagających dostępu do pamięci. Dlatego procesory graficzne są najczęściej używane do przetwarzania obrazów, grafiki komputerowej oraz czasochłonnych i skomplikowanych arytmetycznie algorytmów.[36][8]



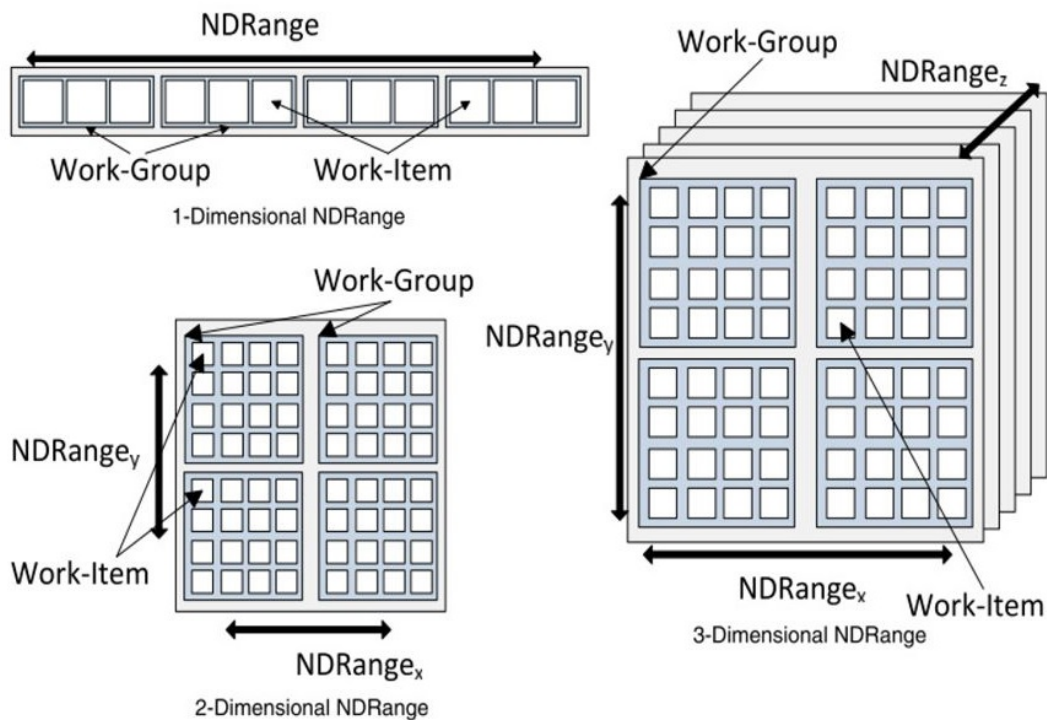
Rysunek 3.10: Porównanie budowy CPU względem GPU[44]

### 3.3.2 Biblioteka OpenCL

Biblioteka OpenCL została stworzona do ułatwienia tworzenia programów dla systemów heterogenicznych. Wykorzystuje trend stosowania rozwiązań wielordzeniowych w architekturze nowoczesnych procesorów. OpenCL wspiera tworzenie aplikacji dla wielordzeniowych mikroprocesorów, kart graficznych, układów FPGA i DSP.[8] OpenCL jest to API w języku C z dodatkowymi powiązaniem w językach C++, .NET, Java i Python. Kod przeznaczony dla docelowego urządzenia(CPU lub GPU) jest napisany w języku OpenCL C. Jest to ograniczona wersja języka C w standardzie C99.

Specyfikacja OpenCL jest podzielona na cztery modele[8][17][40]:

1. Model platformy(*Platform model*): jeden procesor(*Host*) zarządza pracą pozostałych urządzeń(*Device*) wykonujących programy napisane w OpenCL C, zwane kernelami(*kernels*). Urządzenie jest podzielone na wiele jednostek wykonawczych(*compute units*), gdzie każde z nich składa się z wielu elementów przetwarzania(*processing elements*). Są one odpowiedzialne za przeprowadzanie wszystkich obliczeń.
2. Model wykonawczy(*Execution model*): określa konfigurację środowiska hosta oraz ustawień kerneli.
3. Model pamięci(*Memory model*): definiuje hierarchię pamięci używaną przez kerneli
4. Model programistyczny(*programming model*): określa dwa rodzaje współbieżności - zadaniową(**TLP**) i danych(**DLP**), który jest uważany za preferowany dla aplikacji OpenCL.



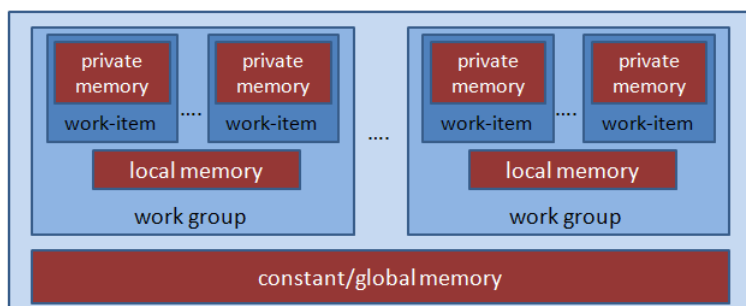
Rysunek 3.11: Przestrzeń indeksowa OpenCL [17]

Wykonywanie programu opartego o OpenCL składa się z dwóch części - instrukcji hosta oraz kerneli przetwarzanych urządzenia OpenCL. Host definiuje abstrakcyjny kontener zwany kontekstem(*context*), który służy do interakcji z urządzeniami OpenCL, zarządzania pamięcią i nadzorowaniem programów i kerneli wykonywanych przez każde urządzenie. Po zdefiniowaniu i inicjalizacji kernela przez hosta tworzona jest N-wymiarowa przestrzeń indeksowa. Instancja programu jest wykonywana dla każdego z elementów przestrzeni, które nazywane są elementami roboczymi(*work-item*). Są one zorganizowane w grupy

robocze(*work-groups*), gdzie każda grupa posiada unikalne ID. Każdy element roboczy posiada ID globalne względem całej przestrzeni indeksowej(zwanej też *NDRange*) oraz lokalne w ramach swojej grupy roboczej. Przestrzeń indeksowa może mieć maksymalnie trzy wymiary(patrz rys.3.11).[8][17][40]

W ramach kernela OpenCL, dla każdego elementu roboczego dostępne są(patrz rys.3.12)

- pamięć globalna(*global memory*): wspólna dla wszystkich elementów przestrzeni indeksowej.
- pamięć stała(*constant memory*): po inicjalizacji przez hosta nie zmienia się w czasie wykonywania programu. Możliwe jest tylko odczytywanie danych.
- pamięć lokalna(*local memory*): indywidualna dla każdej grupy roboczej.
- pamięć prywatna(*private memory*): osobna dla każdego elementu roboczego, nie jest widoczna dla pozostałych



Rysunek 3.12: Model pamięci urządzenia OpenCL [17]

Typowa aplikacja wykorzystująca bibliotekę OpenCL składa się z(patrz rys.3.6 - 3.11):

1. Pobrania informacji na temat platformy(wersja API, profil)

Listing 3.6: OpenCL Platform[40]

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <CL/cl.h>
4
5  #define VECTOR_SIZE 1024
6
7  //OpenCL kernel which is run for every work item created.
8  const char *saxpy_kernel =
9  "__kernel void saxpy_kernel(float alpha, \n"
10 "                                __global float *A, \n"
11 "                                __global float *B, \n"
12 "                                __global float *C) \n"
13 "{ \n"
14 "    //Get the index of the work-item \n"
15 "    int index = get_global_id(0); \n"
16 "    C[index] = (alpha * A[index]) + B[index]; \n"
17 "}" \n";
18
19 int main(void) {
20     int i;
21     // Allocate space for vectors A, B and C
22     float alpha = 2.0;
23     float *A = (float*) malloc(sizeof(float)*VECTOR_SIZE);
24     float *B = (float*) malloc(sizeof(float)*VECTOR_SIZE);
25     float *C = (float*) malloc(sizeof(float)*VECTOR_SIZE);

```



```

26
27     for(i = 0; i < VECTOR_SIZE; i++)
28     {
29         A[i] = i;
30         B[i] = VECTOR_SIZE - i;
31         C[i] = 0;
32     }
33     // Get platform and device information
34     cl_platform_id * platforms = NULL;
35     cl_uint num_platforms;
36
37     //Set up the Platform
38     cl_int clStatus = clGetPlatformIDs(0, NULL, &num_platforms);
39     platforms = (cl_platform_id *)
40     malloc(sizeof(cl_platform_id)*num_platforms);
41     clStatus = clGetPlatformIDs(num_platforms, platforms, NULL);

```

## 2. Określenia listy dostępnych urządzeń i wybrania pożądaných

Listing 3.7: OpenCL Device [40]

```

42
43     //Get the devices list and choose the device you want to run on
44     cl_device_id *device_list = NULL;
45     cl_uint num_devices;
46     clStatus = clGetDeviceIDs( platforms[0], CL_DEVICE_TYPE_GPU, 0,
47     NULL, &num_devices);
48     device_list = (cl_device_id *)
49     malloc(sizeof(cl_device_id)*num_devices);
50
51     clStatus = clGetDeviceIDs( platforms[0],
52     CL_DEVICE_TYPE_GPU, num_devices, device_list, NULL);

```

## 3. Stworzenia kontekstu dla urządzenia do zarządzania pamięcią, kolejkami rozkazów oraz kernelami

## 4. Inicjalizacji kolejki rozkazów(*command queue*)

Listing 3.8: OpenCL Context i OpenCL CommandQueue [40]

```

54     // Create one OpenCL context for each device in the platform
55     cl_context context;
56     context = clCreateContext( NULL, num_devices, device_list,
57     NULL, NULL, &clStatus);
58
59     // Create a command queue
60     cl_command_queue command_queue = clCreateCommandQueue(
61     context, device_list[0], 0, &clStatus);

```

## 5. Stworzenia buforów pamięci wykorzystywanych do wymiany danych pomiędzy hostem a urządzeniem OpenCL.

Listing 3.9: OpenCL Buffer [40]

```

63     // Create memory buffers on the device for each vector
64     cl_mem A_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY,
65     VECTOR_SIZE * sizeof(float), NULL, &clStatus);
66     cl_mem B_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY,
67     VECTOR_SIZE * sizeof(float), NULL, &clStatus);
68     cl_mem C_clmem = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
69     VECTOR_SIZE * sizeof(float), NULL, &clStatus);

```

```

70
71 // Copy the Buffer A and B to the device
72 clStatus = clEnqueueWriteBuffer(command_queue, A_clmem,
73 CL_TRUE, 0, VECTOR_SIZE * sizeof(float),
74 A, 0, NULL, NULL);
75 clStatus = clEnqueueWriteBuffer(command_queue, B_clmem,
76 CL_TRUE, 0, VECTOR_SIZE * sizeof(float),
77 B, 0, NULL, NULL);

```

6. Zbudowanie programu w OpenCL C i stworzenie kernela

7. Powiązanie buforów z argumentami kernela

8. Wykonanie kernela na urządzeniu

Listing 3.10: OpenCL Context [40]

```

79 // Create a program from the kernel source
80 cl_program program = clCreateProgramWithSource(context, 1,
81 (const char **)&saxpy_kernel, NULL, &clStatus);
82
83 // Build the program
84 clStatus = clBuildProgram(program, 1, device_list, NULL,
85 NULL, NULL);
86
87 // Create the OpenCL kernel
88 cl_kernel kernel = clCreateKernel(program, "saxpy_kernel",
89 &clStatus);
90
91 // Set the arguments of the kernel
92 clStatus = clSetKernelArg(kernel, 0, sizeof(float),
93 (void *)&alpha);
94 clStatus = clSetKernelArg(kernel, 1, sizeof(cl_mem),
95 (void *)&A_clmem);
96 clStatus = clSetKernelArg(kernel, 2, sizeof(cl_mem),
97 (void *)&B_clmem);
98 clStatus = clSetKernelArg(kernel, 3, sizeof(cl_mem),
99 (void *)&C_clmem);
100
101 // Execute the OpenCL kernel on the list
102 size_t global_size = VECTOR_SIZE; // Process the entire lists
103 size_t local_size = 64; // Process one item at a time
104 clStatus = clEnqueueNDRangeKernel(command_queue, kernel, 1,
105 NULL, &global_size, &local_size, 0, NULL, NULL);

```

9. Odczytanie przez hosta wyników z buforów pamięci

10. Zakończenie programu urządzenia i zwolnienie pamięci dla wszystkich obiektów i buforów.

Listing 3.11: OpenCL Context [40]

```

107 // Read the cl memory C_clmem on device to the host variable C
108 clStatus = clEnqueueReadBuffer(command_queue, C_clmem,
109 CL_TRUE, 0, VECTOR_SIZE * sizeof(float), C, 0, NULL, NULL);
110
111 // Clean up and wait for all the comands to complete.
112 clStatus = clFlush(command_queue);
113 clStatus = clFinish(command_queue);
114
115 // Display the result to the screen
116 for(i = 0; i < VECTOR_SIZE; i++)

```

```
117         printf("%f * %f + %f = %f\n", alpha, A[i], B[i], C[i]);
118
119     // Finally release all OpenCL allocated objects and
120     host buffers.
121     clStatus = clReleaseKernel(kernel);
122     clStatus = clReleaseProgram(program);
123     clStatus = clReleaseMemObject(A_clmem);
124     clStatus = clReleaseMemObject(B_clmem);
125     clStatus = clReleaseMemObject(C_clmem);
126     clStatus = clReleaseCommandQueue(command_queue);
127     clStatus = clReleaseContext(context);
128     free(A);
129     free(B);
130     free(C);
131     free(platforms);
132     free(device_list);
133     return 0;
134 }
```

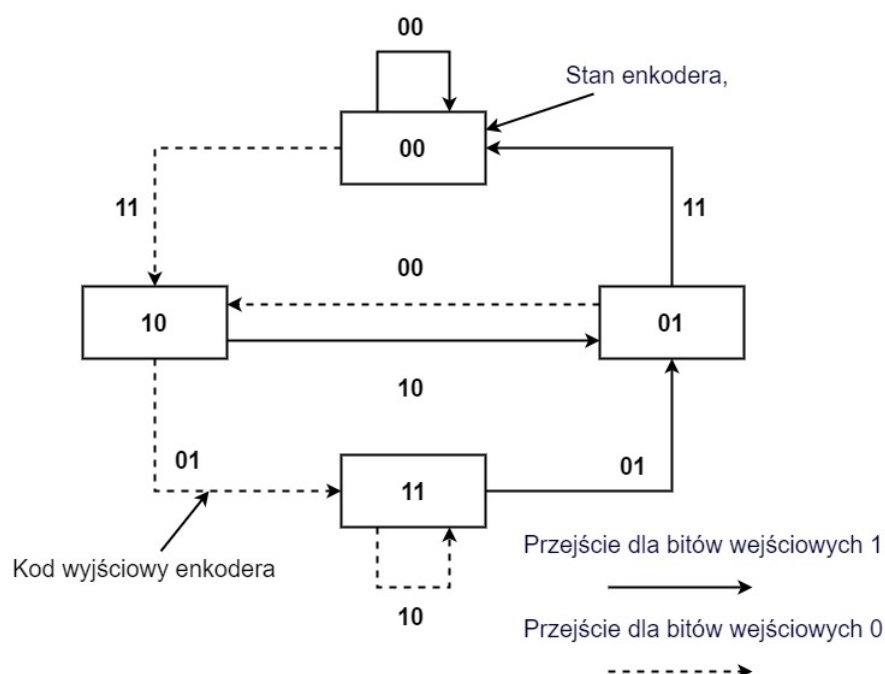
## Rozdział 4

# Algorytm Viterbiego

### 4.1 Opis działania i zastosowania

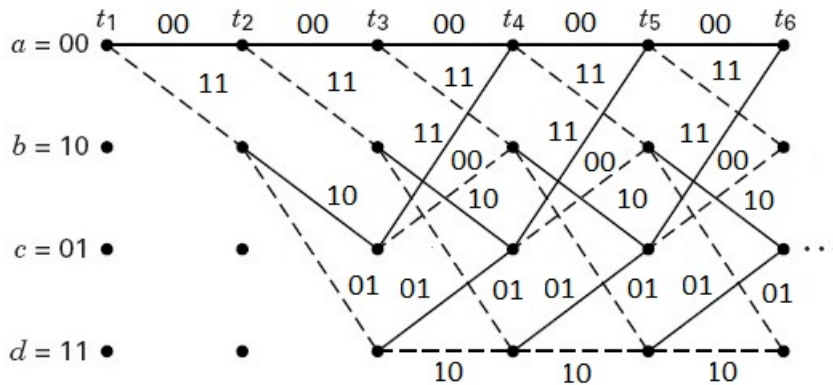
Algorytm Viterbiego został stworzony i przeanalizowany przez A.J. Viterbiego w 1967r. Jego zadaniem było dekodowanie kodów splotowych. Później odkryto że posiada cechy programowania dynamicznego i wykorzystuje maksymalne prawdopodobieństwo do określenia optymalnego zestawu tranzycji pomiędzy stanami. Algorytm został oryginalnie opracowany z myślą o zastosowaniach w telekomunikacji, ale znalazł zastosowanie w innych dziedzinach, m.in. w przetwarzaniu obrazów, lokalizacji i rozpoznawaniu obiektów.[31]

Główne zastosowanie algorytmu Viterbiego polega na dekodowaniu informacji zakodowanych przy pomocy kodów splotowych. Sekwencja kodowana  $m = m_1, m_2, \dots, m_n$ , gdzie  $m_i$  reprezentuje pojedynczy bit informacji, a indeks oznacza kolejność przesyłania. Encoder splotowy przekształca informację wejściową w zakodowaną sekwencję  $U = G(m)$ . Wykorzystuje do tego rejestr przesuwający, sumatory modulo-2 oraz wielomiany generujące (*generator polynomials*) określające związek sumatorów z rejestrem.[42][31]



Rysunek 4.1: Diagram stanów, dla enkodera o 3 bitowym rejestrze przesuwającym i dwóch sumatorach modulo-2 z wielomianami  $g_1 = 111$ ,  $g_2 = 101$  [42]

Enkoder spłotowy należy do klasy urządzeń zwanych automatami skończonymi (*finite-state machines*), które zachowują informację o poprzednich sygnałach. Jego działanie można zaprezentować w postaci diagramu stanów (*state diagram*). Bieżący stan reprezentują pierwsze  $K - 1$  bity rejestru o pojemności  $K$ , a przejścia pomiędzy stanami są określane przez kody wyjściowe. Dla enkodera z 3 bitowym rejestrem przesuwającym i 2 bitowym wyjściem, stany są określone jako kolejno 00, 01, 10, 11. Diagram 4.1 przedstawia omawiany typ enkodera, ze wszystkimi możliwymi tranzycjami. Widoczne są dwa rodzaje przejść pomiędzy stanami - dla bitu wejściowego będącego jedynką oraz dla zera. Do wizualizacji kolejnych tranzycji w czasie gdy ładowane są nowe dane do rejestru przesuwającego, używane są diagramy kratowe (*trellis diagram*). [47][42][31] Kolumny diagramu oznaczają stany rejestru, wiersze kolejne momenty czasu  $t_1, t_2, \dots, t_n$ . Linie łączące ze są węzły diagramu oznaczają tranzycje, które są wyzwalane pojawieniem się nowego bitu sygnału wejściowego w rejestrze enkodera (patrz rys. 4.2).



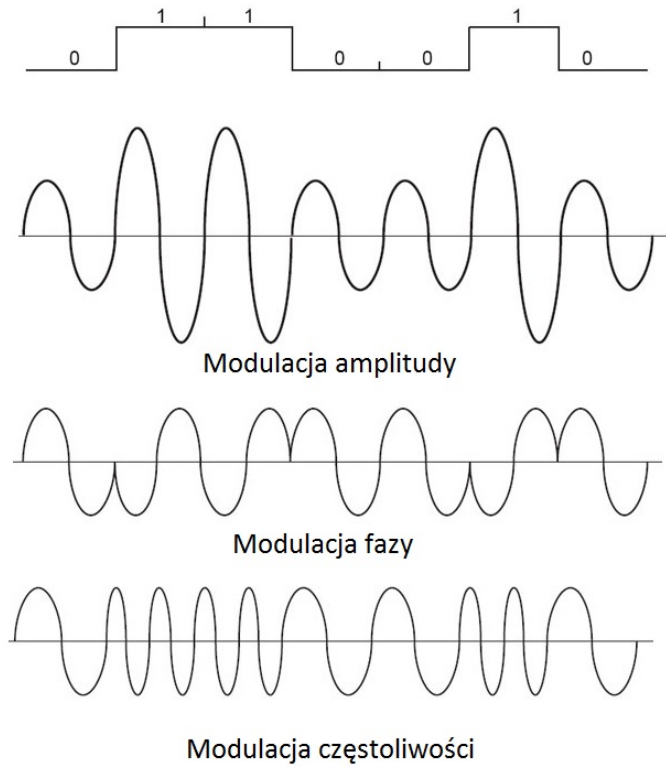
Rysunek 4.2: Diagram kratowy, dla enkodera o 3 bitowym rejestrze przesuwającym i dwóch sumatorach modulo-2 z wielomianami  $g_1 = 111, g_2 = 101$  [42]

Po zakodowaniu informacji jest ona następnie poddawana modulacji - nałożenia wejściowego sygnału na analogowy sygnał nośny modulatora. [42][38] Sygnał cyfrowy ze względu na małą częstotliwość jest podatny na zakłócenia w trakcie transferu. Stosując modulację sygnałem analogowym o wyższej częstotliwości możliwe jest przesyłanie informacji na większe odległości oraz zmniejszenie wpływu zakłóceń. Sygnałem nośnym najczęściej są sygnały sinusoidalne, np. w postaci [22][38]:

$$f_c(t) = A_c \cos(\omega t + \theta) \quad (4.1)$$

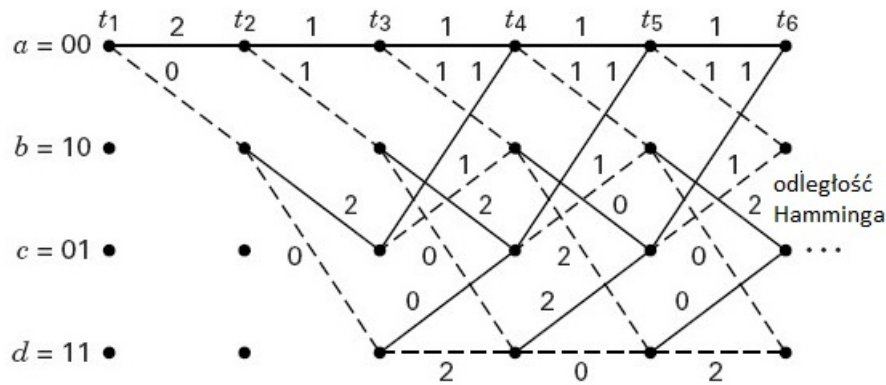
Na podstawie równania 4.1 można stwierdzić że występują trzy parametry, na które można wpłynąć do modyfikacji modulowanego sygnału - amplituda  $A_c$ , częstotliwość  $\omega$  oraz faza  $\theta$ . Dlatego wyróżniamy trzy rodzaje modulacji (patrz rys. 4.3):

- modulacja częstotliwości - jest osiągana poprzez zmianę częstotliwości sygnału w zależności, czy w danym momencie występuje zmiana z 0 na 1 (zwiększenie częstotliwości), czy z 1 na 0 (zmniejszenie częstotliwości).
- modulacja amplitudy - polega na zmodyfikowaniu amplitudy sygnału nośnego przez amplitudę sygnału zakodowanego.
- modulacja fazy - różnica pomiędzy 0 i 1 w sygnale wejściowym jest reprezentowana przez zmianę startu sygnału sinusoidalnego nośnika, np.  $0^\circ$  dla 0 i  $180^\circ$  dla 1.



Rysunek 4.3: Rodzaje modulacji sygnału cyfrowego [42][38]

Odbiornik dostaje zakodowaną, modulowaną i zakłóconą informację. W pierwszej fazie odczytywania przesłanego sygnału niezbędna jest jego demodulacja. Polega ona na odzyskaniu sygnału z wyjścia enkodera przed modulacją. Dla sygnału modulowanego amplitudowo **AM** demodulacja może być przeprowadzona poprzez użycie detektora diodowego oraz filtracji niepożądanych wysokich częstotliwości.[39] W przypadku demodulacji częstotliwościowej **FM** i fazy **PM**, wykorzystywane są detektory **PLL** (*Phase Locked Loop* - pętla synchronizacji fazy)[35].



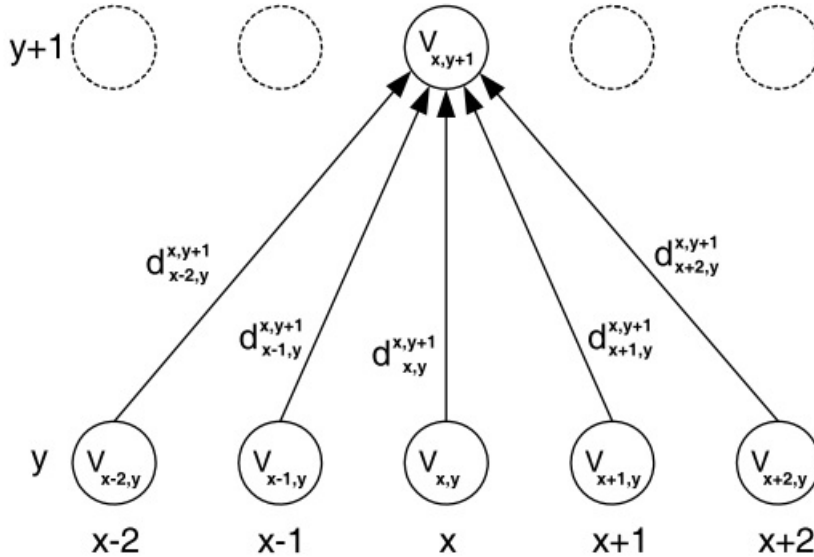
Rysunek 4.4: Dekodowanie z wykorzystaniem algorytmu Viterbiego dla sekwencji zakodowanej enkoderem z 3 bitowym rejestrem przesunym i dwoma sumatorami modulo-2 o wielomianach  $g_1 = 111, g_2 = 101$  [42]

W celu odczytania informacji zakodowanej enkoderem splotowym stosowany jest dekodery wykorzystujący algorytm Viterbiego. Wykorzystuje on strukturę diagramu kratowego do obliczenia optymalnej kombinacji przejścia pomiędzy jego stanami (patrz rys.4.5). Do określenia najlepszego wariantu przejścia ze stanu w czasie  $t_i$  do stanu  $t_{i+1}$ , wybierana jest tranzycja o najmniejszej odległości Hamminga[42]

względem fragmentu kodu po demodulacji. Odległość Hamminga określana jest przez liczbę pozycji, na których różnią się dwa ciągi kodu. Jeśli występuje kilka możliwych przejść do danego stanu, wybierane jest te, które ma największe prawdopodobieństwo - najmniejszą odległość Hamminga, a reszta jest odrzucana. Wybór najlepszego zestawu tranzycji dla całego diagramu jest określony przez skumulowaną odległość Hamminga dla końcowego stanu. Na podstawie otrzymanej optymalnej listy przejść odczytywana jest zakodowana informacja .[42][31][32]

## 4.2 Wykrywanie linii na obrazie cyfrowym z wykorzystaniem algorytmu Viterbiego

Algorytm Viterbiego, którego celem jest znalezienie optymalnej ścieżki w diagramie kratowym, może zostać zastosowany w analogiczny sposób do wyznaczenia optymalnego zestawu współrzędnych pikseli na obrazie cyfrowym. W tym wypadku piksele odpowiadają stanom rejestru przesuwającego, a odległość Hamminga wykorzystana do obliczenia najlepszej tranzycji jest zastąpiona funkcją intensywności pikseli. Zgodnie z zaproponowanym rozwiązaniem w [20] dla każdego piksela określone jest najlepsze przejście do następnej kolumny/wiersza.



Rysunek 4.5: Lokalne tranzycje diagramu kratowego dla obrazu cyfrowego[19]

Zakładając szukanie linii pionowej, przetwarzanie obrazu zaczyna się od pierwszego wiersza  $y = 0$ . Dla każdego z pikseli w pierwszym wierszu jest przypisana początkowa wartość akumulowana  $V_{x,0} = 0$ . Dla piksela o współrzędnej  $x, y$  rozpatrywana jest jego intensywność oraz intensywność jego sąsiadów. Współrzędne sąsiednich pikseli są określane przedziałem  $x + g$ , gdzie  $g \in \langle g_l; g_h \rangle$ , np.  $g_l = -2$ ,  $g_h = 2$ , czyli  $g \in \{-2, -1, 0, 1, 2\}$ . Wartość  $V_{x,y+1}$  może być wyznaczana na podstawie równania 4.2[19][29]:

$$V_{x,y+1} = \max(V_{x+g,y} + d_{x+g,y}^{x,y+1}), \quad g \in \{-2, -1, 0, 1, 2\} \quad (4.2)$$

,gdzie  $V_{x+g,y}$  - skumulowana dotychczasowa intensywność dla współrzędnych piksela  $(x + g, y)$ ,  $d_{x+g,y}^{x,y+1}$  - intensywność piksela o współrzędnych  $(x + g, y)$ ,  $V_{x,y+1}$  - wynikowa skumulowana intensywność dla piksela o współrzędnych  $(x, y + 1)$ .

Wartości  $g$  określające najlepsze lokalną tranzycje są zapisywane w tablicy indeksów  $L$  zgodnie z równaniem 4.3[19][29]:

$$L_y^{x,y+1} = \underset{g}{\operatorname{argmax}}(V_{x+g,y} + d_{x+g,y}^{x,y+1}), \quad g \in \{-2, -1, 0, 1, 2\} \quad (4.3)$$

Indeksy są obliczane dla wszystkich rozpatrywanych wierszy. Po wyznaczeniu tablicy  $L$  sprawdzany jest ostatni wiersz  $y_{max}$  w celu określenia kolumny z największą skumulowaną intensywnością  $x_{max}$  4.4:

$$x_{y_{max}} = \max_x(V_{x,y_{max}}) \quad (4.4)$$

Wsteczne przetwarzanie tablicy indeksów  $L$  pozwala wyznaczyć pozycję fragmentu linii dla pierwszej kolumny  $y = 1$  4.5:

$$x_{y-1} = x_y + L_{y-1}^{x,y}, \quad y = y_{max}, \dots, y_{start}+1 \quad (4.5)$$

Do obliczenia wyników dla następnych kolumn zwiększa się  $y_{start}$  i od nowa oblicza się tablice  $V$  i  $L$ .

## 4.3 Implementacja detekcji linii języku C++

Tworząc aplikację wykorzystującą algorytm Viterbiego do lokalizacji linii na obrazie cyfrowym skorzystano z nowych funkcjonalności standardu C++11. Zdjęcia na, których szukano linii były wczytywane używając funkcji biblioteki `CImg`. Reprezentowane przez obiekty `Cimg<T>`, dane pikseli zdjęcia były kopiowane do dynamicznie zaalokowanych tablic przypisanych do obiektów `std::unique_ptr`. Skorzystano inteligentnego wskaźnika `std::unique_ptr`, ze względu na funkcję automatycznego zwalniania zaalokowanych zasobów po wywołaniu jego destruktor `std::unique_ptr`. Wszystkie warianty implementacji detekcji linii zostały zawarte w klasie `Viterbi`, jako jej metody.

### 4.3.1 Wersja szeregową

Pierwsza podstawowa implementacja algorytmu detekcji linii, w wykonuje wszystkie operacje w sposób sekwencyjny w ramach tego samego wątku. Została stworzona jako punkt odniesienia szybkości wykonania względem później omawianych algorytmów współbieżnych.

Wersja szeregową algorytmu Viterbiego do detekcji linii jest metodą klasy `Viterbi` o nazwie `viterbiLineDetect`. Przyjmuje trzy argumenty wejściowe:

1. `std::vector<unsigned int> &line_x` - referencja kontenera, do którego zapisywane są współrzędne szukanej linii
2. `int g_low` i `int g_high` : określają zakres rozpatrywanego sąsiedztwa każdego piksela przy wyznaczaniu macierz  $V$  (patrz równanie 4.2)

Informacje o intensywności pikseli obrazu pobierane są używając składowej klasy `Viterbi` `const unsigned char *m_img`.



Kod we wnętrzu funkcji dzieli się na:

1. Inicjalizację (patrz rys. 4.1)
2. Pętlę przejścia przez wszystkie kolumny obrazu, w której dla każdej z nich wykonywane są kroki przedstawione we wzorach 4.2 do 4.5. (patrz rys. 4.2)

Listing 4.1: Inicjalizacja zmiennych

```
222 int Viterbi::viterbiLineDetect(std::vector<unsigned int> &line_x, int g_low, int
    g_high)
223 {
224     if (m_img == 0 && m_img_height > 0 && m_img_width > 0)
225     {
226         return 1;
227     }
228     //allocate array for viterbi algorithm
229     std::vector<uint32_t> L(m_img_height * m_img_width, 0);
230     std::vector<uint32_t> V(m_img_height * m_img_width, 0);
231
232     uint32_t P_max = 0;
233     uint32_t x_max = 0;
234     std::vector<uint32_t> x_cord(m_img_width, 0);
235     uint32_t max_val = 0;
236     size_t i = 0;
237     unsigned char pixel_value = 0;
238     while (i < (m_img_width - 1))
```

Listing 4.2: Główna pętla programu

```
238     while (i < (m_img_width - 1))
239     {
240         // init first column with zeros
241         for (size_t m = 0; m < m_img_height; m++)
242         {
243             V[(m * m_img_width) + i] = 0;
244         }
245         for (size_t n = i; n < (m_img_width - 1); n++)
246         {
247             for (int j = 0; j < m_img_height; j++)
248             {
249                 max_val = 0;
250                 for (int g = g_low; g <= g_high; g++)
251                 {
252                     if ((j + g) > (int)(m_img_height - 1))
253                     {
254                         break;
255                     }
256                     if (j + g < 0)
257                     {
258                         continue;
259                     }
260                     int curr_id = j + g;
261                     pixel_value = m_img[((curr_id) * m_img_width) + n];
262                     if ((pixel_value + V[(m_img_width * curr_id) + n]) > max_val)
263                     {
264                         max_val = pixel_value + V[(m_img_width * curr_id) + n];
265                         L[(j * m_img_width) + n] = g;
266                     }
267                 }
268                 V[(j * m_img_width) + (n + 1)] = max_val;
```

```

269     }
270 }
271 //find biggest cost value in last column
272 for (size_t j = 0; j < m_img_height; j++)
273 {
274     if (V[(j * m_img_width) + (m_img_width - 1)] > P_max)
275     {
276         P_max = V[(j * m_img_width) + (m_img_width - 1)];
277         x_max = j;
278     }
279 }
280 //backwards phase - retrace the path
281 x_cord[(m_img_width - 1)] = x_max;
282 for (size_t n = (m_img_width - 1); n > i; n--)
283 {
284     x_cord[n - 1] = x_cord[n] + L[(x_cord[n] * m_img_width) + (n - 1)];
285 }
286 // save only last pixel position
287 line_x[i] = x_cord[i];
288 P_max = 0;
289 x_max = 0;
290 ++i;
291 }

```

### 4.3.2 Wersja równoległa - C++11

Na podstawie wcześniej opisanego algorytmu detekcji linii oraz opisanej szeregowej implementacji można stwierdzić, że możliwe jest oddzielne przetwarzanie kodu dla każdej z kolumn reprezentowanych licznikiem i pętli z listingu 4.2. Pierwszym z wariantów, który wykorzystuje podaną cechę algorytmu jest implementacja wielowątkowa. Korzysta ona z funkcji i obiektów nagłówka `<thread>`, w tym wypadku używany jest obiekt `std::future` do przechowywania rezultatów wykonania wątków. Ponadto do uruchamiania nowych wątków stosowana jest funkcja `std::async`, która uruchamia dodatkowe wątki (patrz rozdział 3.2.3).

Listing 4.3: Wielowątkowa impleme

```

390 int Viterbi::launchViterbiMultiThread(std::vector<unsigned int>& line_x, int g_low,
391 int g_high)
392 {
393     uint32_t to_process = m_img_width - 1;
394     uint32_t start_col = 0;
395     uint32_t idx = 0;
396     uint8_t num_of_threads = std::thread::hardware_concurrency();
397     std::vector<unsigned int> line(num_of_threads);
398     std::vector<std::future<unsigned int>> viterbiThreads(num_of_threads);
399     while (to_process > 0)
400     {
401         uint8_t launched_threads = 0;
402         for (uint8_t i = 0; i < num_of_threads; i++)
403         {
404             if (start_col < m_img_width - 1)
405             {
406                 viterbiThreads[i] = (std::async(launch::async, &Viterbi::
407 viterbiMultiThread, this, g_low, g_high, start_col));
408                 ++start_col;
409                 ++launched_threads;
410                 --to_process;
411             }
412             else

```

```

411         {
412             break;
413         }
414     }
415
416     for (uint8_t i = 0; i < launched_threads; i++)
417     {
418         line_x[idx] = viterbiThreads[i].get();
419         ++idx;
420     }
421 }
422 line_x[m_img.width - 1] = line_x[m_img.width - 2];
423 return 0;

```

W celu właściwego wykorzystania zasobów procesora stosowana jest funkcja `std::thread::hardware_concurrency()` do określenia rekomendowanej liczby jednocześnie uruchomionych wątków. Cały obraz jest przetwarzany fragmentami po `num_of_threads` wątków na raz, gdzie dla każdego z nich przypisana jest inna kolumna startowa  $y_{start}$  (patrz równanie 4.5). Po tym jak wszystkie wątki wykonają swoje obliczenia wynik jest zapisywany i zaczyna się nowa iteracja, w której rozpatrywany jest następny zestaw kolumn obrazu. Pętla będzie się wykonywać dopóki nie zostaną przetworzone wszystkie kolumny, co reprezentowane jest przez wartość licznika `to_process`.

### 4.3.3 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[48] . . .	4
2.2	Przykład obrazów używanych w testowaniu pozycji i orientacji elementów[13] . . . . .	5
2.3	Przykład wizyjnej identyfikacji[13] . . . . .	5
2.4	Procesor <i>QorlQ Layerscape LS1028</i> do aplikacji przemysłowych firmy NXP, wyposażony w dwa rdzenie ARMv8[34] . . . . .	6
2.5	Struktura mikrokontrolera <i>STM32F401CC</i> [43] . . . . .	7
3.1	Schemat cyklu pobierania i wykonania instrukcji przez CPU [44] . . . . .	8
3.2	Hierarchia pamięci[44] . . . . .	9
3.3	Trójpoziomowa organizacja pamięci cache [44] . . . . .	10
3.4	Przykład 4-fazowego potoku(pipeline) procesora [45] . . . . .	10
3.5	Intel Core i7, przykład wielordzeniowego procesora wykorzystującego model <b>SMT</b> [44] . .	11
3.6	Rodzaje implementacji <b>TLP</b> . . . . .	12
3.7	Wykorzystanie <b>pthread_join</b> do synchronizacji wątków[6] . . . . .	14
3.8	Model tworzenia wątków w OpenMP[5] . . . . .	17
3.9	Schemat rdzenia procesora wykorzystującego architekturę SIMD [41] . . . . .	22
3.10	Porównanie budowy CPU względem GPU[44] . . . . .	22
3.11	Przestrzeń indeksowa OpenCL [17] . . . . .	23
3.12	Model pamięci urządzenia OpenCL [17] . . . . .	24
4.1	Diagram stanów, dla enkodera o 3 bitowym rejestrze przesuwным i dwóch sumatorach modulo-2 z wielomianami $g_1 = 111, g_2 = 101$ [42] . . . . .	28
4.2	Diagram kratowy, dla enkodera o 3 bitowym rejestrze przesuwным i dwóch sumatorach modulo-2 z wielomianami $g_1 = 111, g_2 = 101$ [42] . . . . .	29
4.3	Rodzaje modulacji sygnału cyfrowego [42][38] . . . . .	30

4.4	Dekodowanie z wykorzystaniem algorytmu Viterbiego dla sekwencji zakodowanej enkodem z 3 bitowym rejestrem przesuwym i dwoma sumatorami modulo-2 o wielomianach $g_1 = 111, g_2 = 101$ [42] . . . . .	30
4.5	Lokalne tranzycje diagramu kratowego dla obrazu cyfrowego[19] . . . . .	31

## Fragmenty kodu

3.1	Przykład tworzenia i uruchamiania wątków . . . . .	14
3.2	Przykład wykorzystanie muteksa do synchronizacji aplikacji wielowątkowej . . . . .	15
3.3	Prosta aplikacja wykorzystująca dyrektywę <code>#pragma omp parallel</code> [37] . . . . .	17
3.4	Podstawowe funkcjonalności <code>std::thread</code> [14] . . . . .	20
3.5	Przykład zastosowania <code>std::future</code> i <code>std::async()</code> [14] . . . . .	21
3.6	OpenCL Platform[40] . . . . .	24
3.7	OpenCL Device [40] . . . . .	25
3.8	OpenCL Context i OpenCL CommandQueue [40] . . . . .	25
3.9	OpenCL Buffer [40] . . . . .	25
3.10	OpenCL Context [40] . . . . .	26
3.11	OpenCL Context [40] . . . . .	26
4.1	Inicjalizacja zmiennych . . . . .	33
4.2	Główna pętla programu . . . . .	33
4.3	Wielowątkowa impleme . . . . .	34



## 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/](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] Lee Howes Benedict Gaster. *Heterogeneous Computing with OpenCL*. Elsevier, 2012. ISBN:9780123877666.
- [9] OpenMP Architecture Review Board. Openmp application program interface. <http://www.openmp.org/wp-content/uploads/spec25.pdf>, 2005.
- [10] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997. ISBN:0201633922.
- [11] Pong P. Chu. *FPGA Prototyping by VHDL examples*. John Wiley and Sons, Inc., 2008. ISBN:9780470185315.
- [12] Cognex. Insight 5000 industrial vision systems. <http://www.cognex.com/productstemplate.aspx?id=13915>.
- [13] Cognex. Introduction to machine vision. [http://www.assemblymag.com/ext/resources/White\\_Papers/Sep16/Introduction-to-Machine-Vision.pdf](http://www.assemblymag.com/ext/resources/White_Papers/Sep16/Introduction-to-Machine-Vision.pdf), 2016.
- [14] C++ Concurrency. C++ reference documentation. <http://en.cppreference.com/>.
- [15] E.R. Davies. *Computer and Machine Vision: Theory, Algorithms, Practicalities*. Elsevier, 225 Wyman Street, Waltham, 02451, USA, 2012. ISBN:9780123869081.
- [16] Standard C++ Foundation. C++11 standard library extensions - concurrency. <https://isocpp.org/wiki/faq/cpp11-library-concurrency>.

- [17] Khronos OpenCL Working Group. The opecl specification. <https://www.khronos.org/registry/OpenCL/specs/openc1-1.1.pdf>, 2011.
- [18] Ian Grout. *Digital Systems Design with FPGAs and CPLDs*. Elsevier, 2008. ISBN:9780750683975.
- [19] Przemysław Mazurek Grzegorz Matczak. Line following with real-time viterbi track-before-detect algorithm. *Przegląd Elektrotechniczny*, 1/2017:69–72, 2017.
- [20] Przemysław Mazurek Grzegorz Matczak. Tracklet-based viterbi track-before-detect algorithm for line following robots. *Proceedings of the 9th International Conference on Computer Recognition Systems*, 2015.
- [21] K Hong. Multi-threaded programming: C++11. [http://www.bogotobogo.com/cplusplus/multithreaded4\\_cplusplus11.php](http://www.bogotobogo.com/cplusplus/multithreaded4_cplusplus11.php).
- [22] Systemy Teleinformatyczne. Katedra Robotyki i Mechatroniki AGH. Modulacja amplitudy. [http://home.agh.edu.pl/~mgi/instrukcje/LAB\\_2\\_Modulacja\\_Amplitudy-student.pdf](http://home.agh.edu.pl/~mgi/instrukcje/LAB_2_Modulacja_Amplitudy-student.pdf).
- [23] National Instruments. Choosing the right camera bus. *NI white papers*, 2016.
- [24] 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>.
- [25] David Patterson John Hennesy. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011. ISBN:9780123838728.
- [26] Mike Houston Katvon Fatahalian. A closer look at gpus. *Communications of the ACM*, 51(10):50–57, 2008.
- [27] Guy Kerens. Multi-threaded programming with posix threads. <http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html>.
- [28] K.N. King. *Język C. Nowoczesne programowanie*. Helion, 2008. ISBN:9788324628056.
- [29] Przemysław Mazurek. Viterbi algorithm for noise line following robots. *Image Processing & Communication Challenges*, 2015.
- [30] Scott Meyers. *Effective Modern C++*. O'Reilly, 2015. ISBN:9781491903995.
- [31] G.R. Nudd M.S. Ryan. The viterbi algorithm. *Warwick Reserch Report*, 1993.
- [32] MIT Digital Communication Systems Lecture Notes. Viterbi decoding of convolutional codes. [https://www.st-andrews.ac.uk/~www\\_pa/Scots\\_Guide/RadCom/part13/page1.html](https://www.st-andrews.ac.uk/~www_pa/Scots_Guide/RadCom/part13/page1.html).
- [33] Nvidia. What is gpu-accelerated computing. <http://www.nvidia.com/object/what-is-gpu-computing.html>.
- [34] NXP. Arm technology-based solutions - nxp microcontrollers and processors. <http://www.nxp.com/products/microcontrollers-and-processors/arm-processors:ARM-ARCHITECTURE>.
- [35] University of St Andrews. Fn & pm demodulation. [https://www.st-andrews.ac.uk/~www\\_pa/Scots\\_Guide/RadCom/part13/page1.html](https://www.st-andrews.ac.uk/~www_pa/Scots_Guide/RadCom/part13/page1.html).
- [36] John Owens. Gpu architecture overview. <http://gpgpu.org/static/s2007/slides/02-gpu-architecture-overview-s07.pdf>.
- [37] Peter S. Pacheco. *An Introduction to Parallel Programming*. Elsevier, 2011. ISBN:9780123742605.

- [38] Instytut Elektrotechniki i Elektroniki Przemysłowej. Zakład Elektrotechniki Teoretycznej i Stosowanej Politechnika Poznańska. Modułacje analogowe i cyfrowe. [http://zetis.iew.put.poznan.pl/Pliki/Lab\\_14/cw4WDT.pdf](http://zetis.iew.put.poznan.pl/Pliki/Lab_14/cw4WDT.pdf).
- [39] Radio-Electronics.com. Amplitude modulation am demodulation. <http://www.radio-electronics.com/info/rf-technology-design/am-reception/amplitude-modulation-detection-demodulation.php>.
- [40] Koushik Bhattacharyya Ravishekhar Banger. *OpenCL Programming by Example*. Pack Publishing, 2012. ISBN:9781849692342.
- [41] Ofer Rosenberg. Introduction to gpu architecture. <http://haifux.org/lectures/267/Introduction-to-GPUs.pdf>.
- [42] Bernard Sklar. *Digital Communications. Fundamentals and Applications*. Prentice Hall, 2001. ISBN:9780130847881.
- [43] ST. Stm32 32-bit arm cortex mcus. <http://www.st.com/en/microcontrollers/stm32-32-bit-arm-cortex-mcus.html?querycriteria=productId=SC1169>.
- [44] William Stallings. *Operating Systems Internals And Design Principles*. Prentice Hall, 2012. ISBN:9780132309981.
- [45] Jon Stokes. *Inside the Machine*. No Starch Press, 2007. ISBN:9781593271046.
- [46] Bjarne Stroustrup. *Język C++.Kompendium wiedzy*. Helion, 2013. ISBN:9788324685301.
- [47] Łukasz Wołowiec. Kod splotowy. <http://ics.p.lodz.pl/~wolowiec/kodowanie/Kod%20Splotowy.pdf>.
- [48] Andy Wilson. Industrial inspection : Line-scan-based vision system tackles color print inspection. *Vision Systems Design*, 2014.
- [49] Joel Yliluoma. Guide into openmp: Easy multithreading programing for c++. <http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html>.