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 8

3 Metody równoległego przetwarzania danych **9**
3.1 Wielowątkowość aplikacji dla języka C/C++ 9
3.1.1 Biblioteka POSIX dla systemów Unix 10
3.1.2 OpenMP - wieloplatformowe API 12
3.1.3 Wielowątkowość w standardzie C++11 12
3.2 Programowanie równoległe z wykorzystaniem GPU 12
3.2.1 Architektura GPU i porównanie względem CPU 12
3.2.2 Biblioteka OpenCL 12

4 Algorytm Viterbiego **13**
4.1 Opis działania i zastosowania 13
4.2 Implementacja w języku C++ 13
4.2.1 Wersja szeregowa 13
4.2.2 Wersja równoległa - C++11 13
4.2.3 Wersja równoległa - OpenCL 13

5 Wyniki badań doświadczalnych implementacji algorytmu Viterbiego **14**
5.1 Porównanie czasu działania dla implementacji szeregowej, wielowątkowej oraz z wykorzystaniem biblioteki OpenCL 14
5.2 Porównanie szybkości algorytmów dla różnych konfiguracji sprzętowych 14

6 Wnioski końcowe **15**

7 Załącznik B **16**

8 Załącznik A **17**

Spis rysunków **18**

9 Bibliografia **20**

Rozdział 1

Streszczenie

To jest streszczenie

Rozdział 2

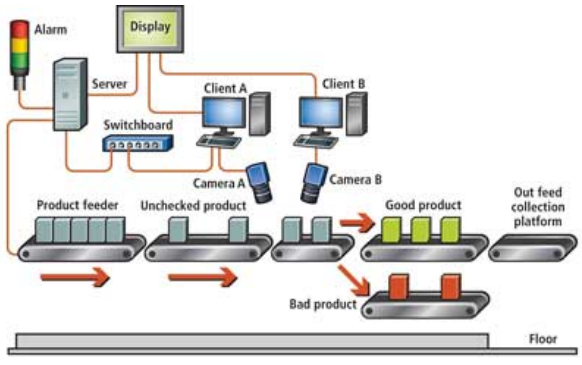
Wstęp

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 [9]:

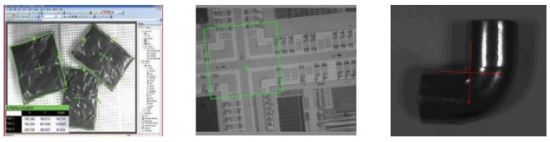
- inspekcja elementów na linii technologicznej
- określanie właściwej orientacji i położenia elementów
- identyfikacja produktów
- pomiary metrologiczne

W automatyce przemysłowej gdzie do zagadnień inspekcji wcześniej niezbędna była ocena wizualna człowieka, obecnie powszechnie stosuje się systemy wizyjne, w których skład wchodzą kamery przemysłowe, czujniki wyzwalające(np. na bazie pozycji) oraz urządzenie odpowiadającego 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.[9][10]



Rysunek 2.1: Przykład zautomatyzowanej linii technologicznej wykorzystującej system wizyjny[23]

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.[9]



Rysunek 2.2: Przykład obrazów używanych w testowaniu pozycji i orientacji elementów[9]

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.[9]

Rysunek 2.3: Przykład wizyjnej identyfikacji[9]

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.

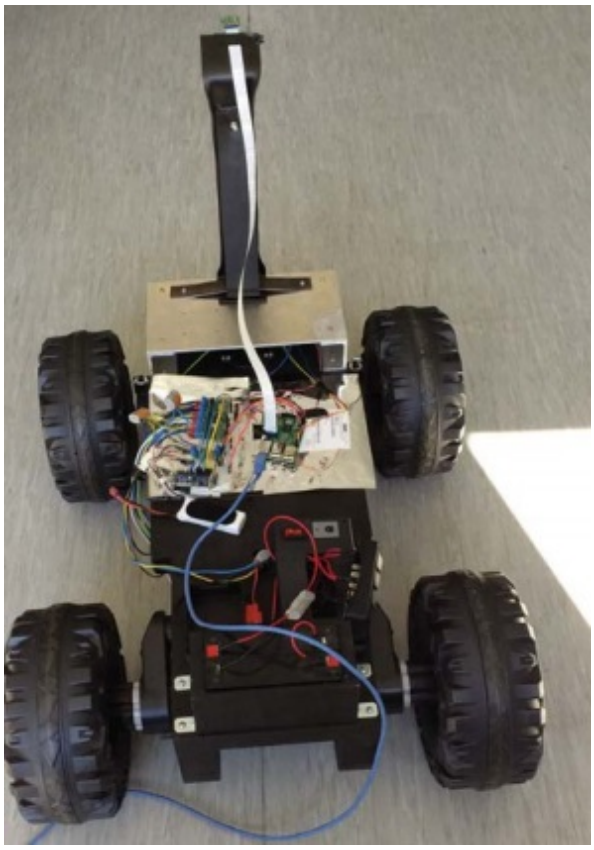
Dla zastosowań przemysłowych. gdzie monitorowane obiekty poruszają się z dużą prędkością, szybkość podjęcia decyzji przez system wizyjny może być wąskim gardłem dla danej gałęzi linii produkcyjnej. Czas na wykonanie decyzji (np. o usunięciu wadliwego produktu z przenośnika taśmowego), składa się na czas 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 [13]. Na podstawie tego można stwierdzić, że główny problem będzie stanowił czas obliczenia sterowania i od niego będzie zależeć szybkość działania systemu wizyjnego[8][5].

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.

[12][1]

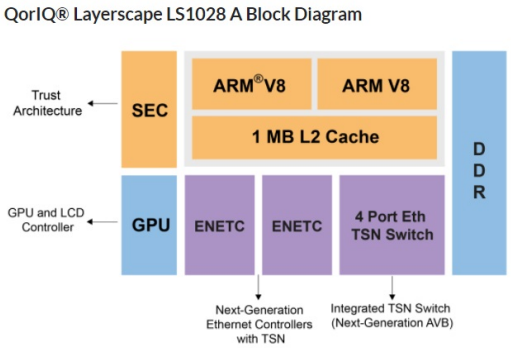
Podobnie jak dla aplikacji przemysłowych odpowiedzialność za wykorzystanie pełnych możliwości układów wykonawczych robota jest szybkość obliczania nowych sterowań. Niezbędne obliczenia mogą być wykonywane bezpośrednio przez urządzenie sterujące silnikami robota, albo z pomocą osobnej



Rysunek 2.4: Robot mobilny do śledzenia linii[12]

stacji, która połączona zdalnie z kontrolerem robota jest odpowiedzialna za przeprowadzanie czasochłonnych obliczeń.

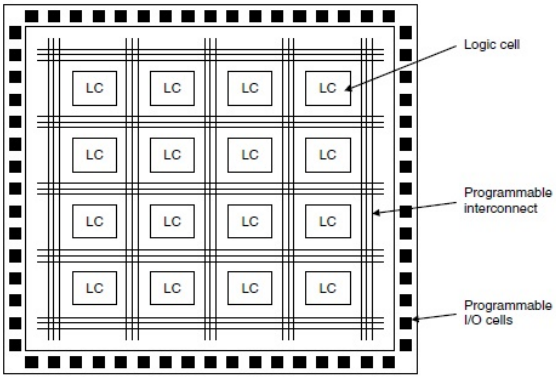
Pierwsze rozwiązanie jest korzystne kiedy nie są wymagane duże rozdzielczości obrazu, skomplikowane i czasochłonne algorytmy przetwarzania obrazu o dużej złożoności obliczeniowej oraz wysokie prędkości ruchu robota, narzucające krótki czas na obliczenia. Stosowane są wtedy najczęściej układy wyposażone mikroprocesory, np. rodzina procesorów ARM oraz x86-64 firmy Intel. Obecne modele są najczęściej wielordzeniowe o taktowaniu nawet powyżej 1GHz [20] [14].



Rysunek 2.5: Procesor *QorlQ Layerscape LS1028* do aplikacji przemysłowych firmy NXP, wyposażony w dwa rdzenie ARMv8[20]

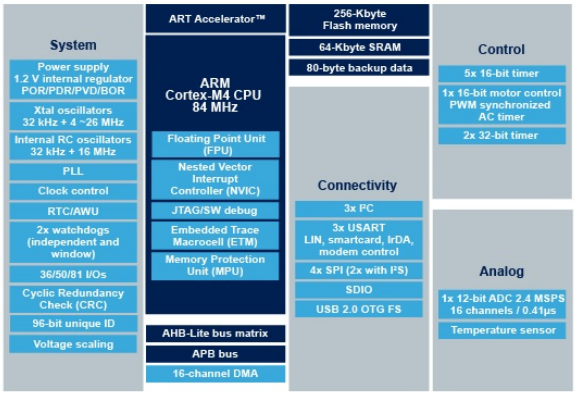
Dla rozwiązań bardziej wymagających pod względem szybkości obliczeń stosowane są układy FPGA, w których logika przetwarzania informacji obrazu jest zaprojektowania w języku HDL(VHDL,

Verilog). Zapewniają one najszybsze prędkości obliczeń ze względu na sprzętową implementację algorytmów. [11][7].



Rysunek 2.6: Architektura układu FPGA[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)[21] [2].



Rysunek 2.7: Structura mikrokontrolera *STM32F401CC*[21]

Zewnętrzna jednostka obliczeniowa pozwala na wykorzystanie możliwości konwencjonalnych wielordzeniowych procesorów dla komputerów PC oraz procesora karty graficznej, który jest wyspecjalizowany w obliczeniach równoległych[19] [15][16]. 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 rożnych wersji napisanego algorytmu. Rozpatrywana będzie implementacja szeregowa i równoległa 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.[15][3]

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.[4][17] 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. [6][22]

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.[6][22]

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.[22][18]

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.[4] Korzystanie z biblioteki Pthreads do tworzenia wątków generuje mniejszy narzut niż tworzenie osobnych procesów do równoległego przetwarzania danych.[4]

Wątek w programie jest reprezentowany poprzez zmienną typu `pthread_t`, najczęściej zdefiniowaną jako zmienna statyczna lub jako struktura, która jest zaalokowana na stercie.[6][4][17] 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`. [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 został stworzone z atrybutem `joinable`, a nie `detached`(odłączony) mogą używać tego rodzaju synchronizacji.[4](patrz 3.1).

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message_function( void *ptr )
{
    char *message;
    //casts void* to readable char* message
    message = (char *)ptr;
    int i;
    for(i = 0; i < 5; i++)
    {
        printf("%s = %d\n", message, i);
        sleep(1);
    }
}

int main()
{
    pthread_t thread1, thread2;
    const char *message1 = "Thread 1";
    const char *message2 = "Thread 2";
    int  iret1, iret2;
    //create and execute thread1
    iret1 = pthread_create(&thread1, NULL, print_message_function, (void*)message1);
    //check if thread1 successfull created
    if(iret1)
    {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret1);
        return iret1;
    }
    //create and execute thread2
    iret2 = pthread_create(&thread2, NULL, print_message_function, (void*)message2);
    //check if thread2 successfull created
    if(iret2)
    {
```

```
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 czytanie 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.[6][17][4] 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`. [6][18] W celu deklaracji muteksa wykorzystywane jest makro `PTHREAD_MUTEX_INITIALIZER`. 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`.

```
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  }
```

Listing 3.2: Przykład wykorzystanie muteksa do synchronizacji aplikacji wielowątkowej

3.1.2 OpenMP - wieloplatformowe API

To jest podrozdział 2 rozdziału 1

3.1.3 Wielowątkowość w standardzie C++11

To jest podrozdział 3 rozdziału 1

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 szeregowa

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[23] . 4
2.2 Przykład obrazów używanych w testowaniu pozycji i orientacji elementów[9] 5
2.3 Przykład wizyjnej identyfikacji[9] 5
2.4 Robot mobilny do śledzenia linii[12] 6
2.5 Procesor *QorlQ Layerscape LS1028* do aplikacji przemysłowych firmy NXP, wyposażony w dwa rdzenie ARMv8[20] 6
2.6 Architektura układu FPGA[11] 7
2.7 Structura mikrokontrolera *STM32F401CC*[21] 7

Listings

3.1 Przykład tworzenia i uruchamiania wątków 10
3.2 Przykład wykorzystanie muteksa do synchronizacji aplikacji wielowątkowej 11

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] Atmel. Atmel avr 8-bit and 32-bit microcontrollers. `http://www.atmel.com/products/microcontrollers/avr/default.aspx`.

[3] Blaise Barney. Introduction to parallel computing. `https://computing.llnl.gov/tutorials/parallel_comp/`.

[4] Blaise Barney. Posix threads programming. `https://computing.llnl.gov/tutorials/pthreads/`.

[5] Basler. Basler camera portfolio. `https://www.baslerweb.com/en/products/cameras/`.

[6] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997. ISBN:0201633922.

[7] Pong P. Chu. *FPGA Prototyping by VHDL examples*. John Wiley and Sons, Inc., 2008. ISBN:9780470185315.

[8] Cognex. Insight 5000 industrial vision systems. `http://www.cognex.com/productstemplate.aspx?id=13915`.

[9] Cognex. Introduction to machine vision. `http://www.assemblymag.com/ext/resources/White_Papers/Sep16/Introduction-to-Machine-Vision.pdf`, 2016.

[10] E.R. Davies. *Computer and Machine Vision: Theory, Algorithms, Practicalities*. Elsevier, 225 WYman Street, Waltham, 02451, USA, 2012. ISBN:9780123869081.

[11] Ian Grout. *Digital Systems Design with FPGAs and CPLDs*. Elsevier, 2008. ISBN:9780750683975.

[12] Przemysław Mazurek Grzegorz Matczak. Line following with real-time viterbi trac-before-detect algorithm. *Przegląd Elektrotechniczny*, 1/2017:69–72, 2017.

[13] National Instruments. Choosing the right camera bus. *NI white papers*, 2016.

[14] 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`.

[15] David Patterson John Hennesy. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011. ISBN:9780123838728.

[16] Mike Houston Katvon Fatahalian. A closer look at gpus. *Communications of the ACM*, 51(10):50–57, 2008.

[17] Guy Kerens. Multi-threaded programming with posix threads. `http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html`.

[18] K.N. King. *Język C. Nowoczesne programowanie*. Helion, 2008. ISBN:9788324628056.

[19] Nvidia. What is gpu-accelerated computing. `http://www.nvidia.com/object/what-is-gpu-computing.html`.

[20] NXP. Arm technology-based solutions - nxp microcontrollers and processors. `http://www.nxp.com/products/microcontrollers-and-processors/arm-processors:ARM-ARCHITECTURE`.

[21] ST. Stm32 32-bit arm cortex mcus. `http://www.st.com/en/microcontrollers/stm32-32-bit-arm-cortex-mcus.html?querycriteria=productId=SC1169`.

[22] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2013. ISBN:9788324685301.

[23] Andy Wilson. Industrial inspection : Line-scan-based vision system tackles color print inspection. *Vision Systems Design*, 2014.