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

Rozdział 1

Streszczenie

To jest streszczenie

Rozdział 2

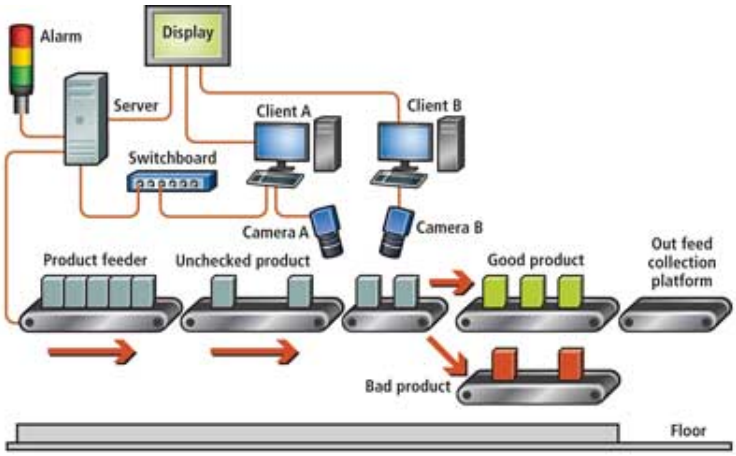
Wstęp

2.1 Przetwarzanie obrazu i jego rola w automatyce przemysłowej

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

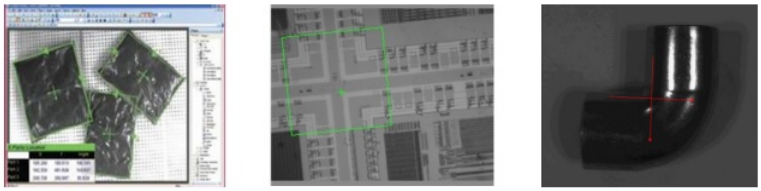
- inspekcja elementów na linii technologicznej
- określanie właściwej orientacji i położenia elementów
- identyfikacja produktów
- pomiary metrologiczne

W automatyce przemysłowej gdzie do zagadnień inspekcji wcześniej niezbędna była ocena wizualna człowieka, obecnie powszechnie stosuje się systemy wizyjne, w których skład 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.[12][14]



Rysunek 2.1: Przykład zautomatyzowanej linii technologicznej wykorzystującej system wizyjny[33]

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



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

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

Rysunek 2.3: Przykład wizyjnej identyfikacji[12]

2.2 Istotność szybkości obliczeń w problemach wizji maszynowej

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

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

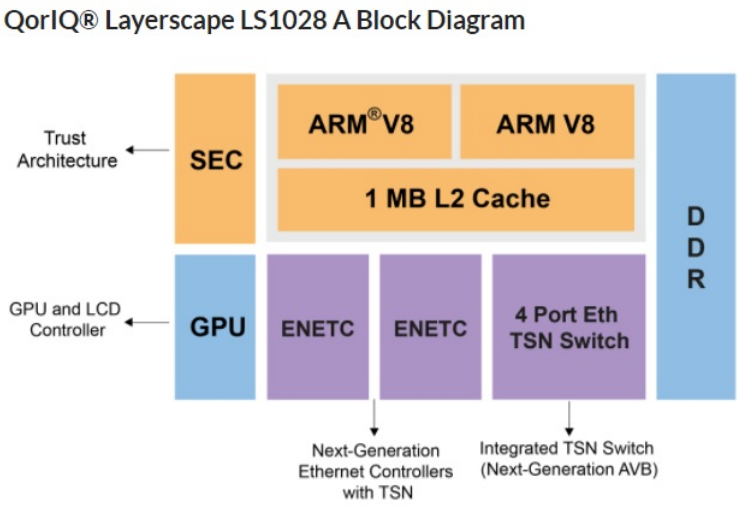
Inną dziedziną gdzie stosowane jest przetwarzanie obrazu w czasie rzeczywistym jest robotyka mobilna, gdzie system wizyjny może odpowiadać za:

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

[17][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.

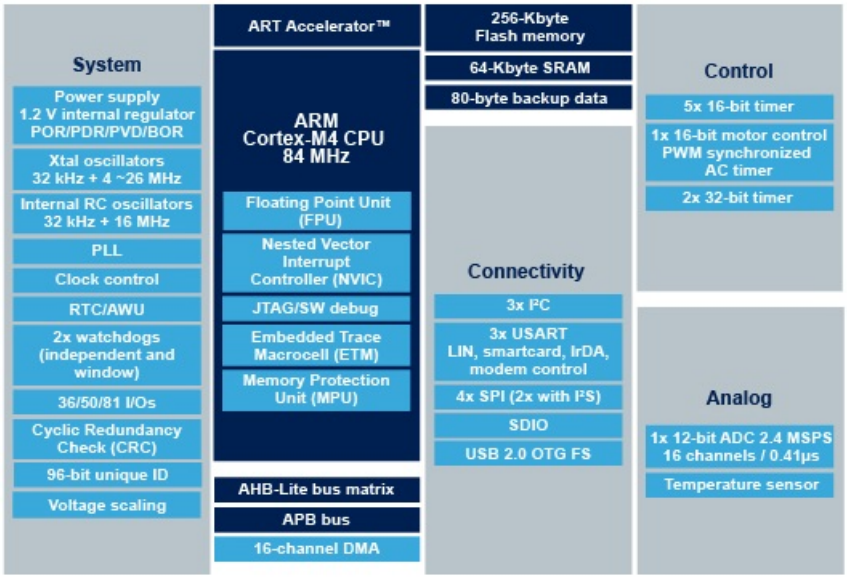
Pierwsze rozwiązanie jest korzystne kiedy nie są wymagane duże rozdzielczości obrazu, skomplikowane i czasochłonne algorytmy przetwarzania obrazu o dużej złożoności obliczeniowej oraz wysokie prędkości ruchu robota, narzucające krótki czas na obliczenia. Stosowane są wtedy najczęściej układy wyposażone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



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

Dla rozwiązań bardziej wymagających pod względem szybkości obliczeń stosowane są układy FPGA, w których logika przetwarzania informacji obrazu jest zaprojektowania w języku HDL(VHDL, Verilog). Zapewniają one najszybsze prędkości obliczeń ze względu na sprzętową implementację algorytmów. [16][10].

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



Rysunek 2.5: Structura mikrokontrolera *STM32F401CC*[29]

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

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

2.3 Cel, zakres i zastosowania pracy

Celem pracy jest implementacja algorytmu Viterbiego w celu wykrywania linii na zaszumionym obrazie cyfrowym oraz analiza porównawcza dla 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.[21][4]

Wątek jest to podproces, który posiada własny stos, zestaw rejestrów, ID, priorytet i wykonuje określony fragment kodu programu. W przeciwieństwie do prawdziwego procesu posiada wspólną pamięć globalną i sterty, dzieloną z innymi wątkami istniejącymi w ramach tego samego procesu. Wątki procesu wykonują się równolegle, dopóki nie potrzebują dostępu do zasobów we wspólnej pamięci.[6][23] Wtedy ze względu na problem błędnego odczytu, bądź zapisu wartości w pamięci kiedy inny wątek ją już nadpisał, może spowodować niewłaściwe działanie programu. Fragmenty kodu gdzie może dojść do tego problemu nazywane są sekcjami krytycznymi. Do zabezpieczania sekcji krytycznych programu stosowane są blokady - muteksy(Mutual exclusions) oraz zmienne warunkowe. Zastosowanie muteksa powoduje, że w danym momencie tylko jeden wątek może wykonywać kod chroniony przez tą blokadę i dopóki nie opuści chronionej sekcji krytycznej inny wątek nie może zacząć jej wykonywać. Zmienne warunkowe stosowane są do sygnalizowania postępu danego wątku, tak aby inny mógł kontynuować wykonywanie operacji sekcji krytycznej. Stosowane razem z blokadami umożliwiają właściwą synchronizację pracy wątków tego samego procesu. [9][32]

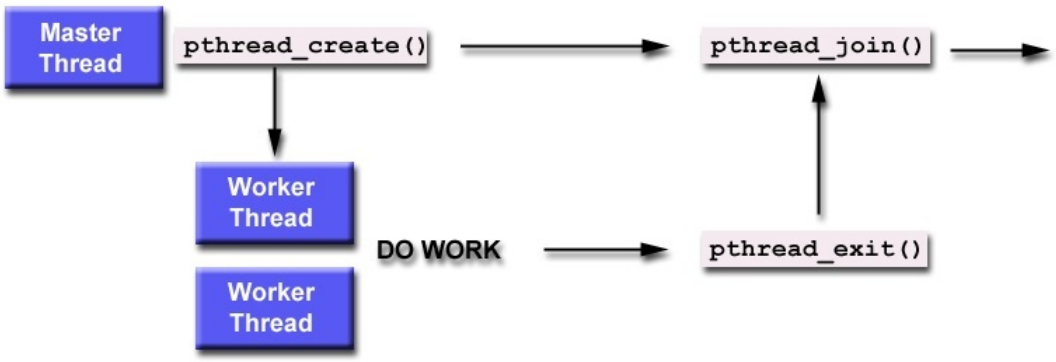
Używanie wielowątkowości w aplikacjach pozwala na wykorzystanie możliwości sprzętowych procesorów wielordzeniowych do obliczeń równoległych. Ponadto wielowątkowy model programowania umożliwia wykonywanie przez proces dalszych działań w czasie czasochłonnych obliczeń, bądź czekania na zdarzenie blokujące - takie jak ,np.sygnał z urządzenia peryferyjnego. Wadą tworzenia dodatkowych wątków w programie jest narzut obliczeniowy związany z ich synchronizacją(omawiany wcześniej problem wyścigu oraz zjawisko zakleszczenia - wątki czekają na siebie nawzajem żeby móc kontynuować obliczenia) oraz dostępem do wspólnego obszaru pamięci.[9][32]

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.[32][24]

3.1.1 Biblioteka POSIX dla systemów Unix

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

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



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

Jedną z podstawowych metod synchronizacji pomiędzy wątkami jest użycie funkcji `pthread_join`, która powoduje zatrzymanie dalszego wykonywania instrukcji dopóki stworzony wątek nie zakończy pracy. Tylko wątki, które został stworzone z atrybutem `joinable`, a nie `detached`(odłączony) mogą używać tego rodzaju synchronizacji.[6](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)
    {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret2);
        return iret2;
    }
    printf("pthread_create() for thread 1 returns: %d\n",iret1);
    printf("pthread_create() for thread 2 returns: %d\n",iret2);
    //wait for threads to finish
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    return 0;
}
```

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.[9][23][6] W bibliotece POSIX threads muteks jest reprezentowany jako zmienna typu `pthread_mutex_t`. W celu posiadania globalnego zasięgu deklarowana jest jako zmienna `static` lub `extern`. [9][24](patrz3.2) W celu deklaracji muteksa wykorzystywane jest makro `PTHREAD_MUTEX_INITIALIZER`(patrz . Jeśli muteks jest używany jako element dynamicznie alokowanej struktury, musi zostać zainicjalizowany wywołaniem funkcji `pthread_mutex_init`. Ponadto musi być w ten sposób inicjalizowany, jeśli nie ma posiadać domyślnych atrybutów. Niezbędne jest po zakończeniu używania muteksa, zwolnienie zaalokowanej pamięci, poprzez wykorzystanie funkcji `pthread_mutex_destroy`.[9]

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

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

void *protected_fun(void *ptr)
{
    char *message;
    pthread_mutex_lock( &mutex1 );
    //casts void* to readable char* message
    message = (char *)ptr;
    int i;
    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

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

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

3.1.2 OpenMP - wieloplatformowe API

Podobnie jak Pthreads, OpenMP jest biblioteką wykorzystującą model współdzielenia pamięci do programowania równoległego. Zawiera wsparcie dla języków C, C++ oraz Fortran. OpenMP wymaga sprecyzowania przez użytkownika odpowiednich akcji, które ma wykonać kompilator, aby program wykonywał się równolegle. [28][8] OpenMP został stworzony przez grupę naukowców i programistów, którzy uważali, że używanie API Pthreads jest skomplikowane dla dużych aplikacji. Zdecydowali się stworzyć standard wyższego poziomu, który w przeciwieństwie do biblioteki Pthreads wymagającej od programisty zdefiniowania funkcji wykonawczej dla każdego wątku, pozwala na określenie dowolnego fragmentu programu, który ma być wykonany równolegle. Wykorzystuje do tego dyrektywy preprocesora znane jako `#pragma`.Używane są do zdefiniowania zachowań kompilatora, które nie są zawarte w podstawowej specyfikacji języka C. Jeśli użyty kompilator nie wspiera dyrektyw `#pragma`, program i tak ma możliwość właściwego działania; wtedy jego fragmenty mające wykonać się równolegle zostaną obsłużone przez jeden wątek. [28][24][8][34] Oprócz zestawu dyrektyw preprocesora, OpenMP składa się z biblioteki funkcji i makr, które wymagają dodania pliku nagłówkowego `<omp.h>` z ich definicjami i prototypami.

Podstawową dyrektywą OpenMP jest dyrektywa `#pragma omp parallel`, za pomocą której określany jest blok kodu mający być wykonany wielowątkowo. Jeśli programista nie sprecyzuje przez ile wątków ma być przetworzony podany fragment programu, zostanie on określony przez system wykonawczy(zazwyczaj po jednym wątku na rdzeń).[28][34](patrz 3.3)

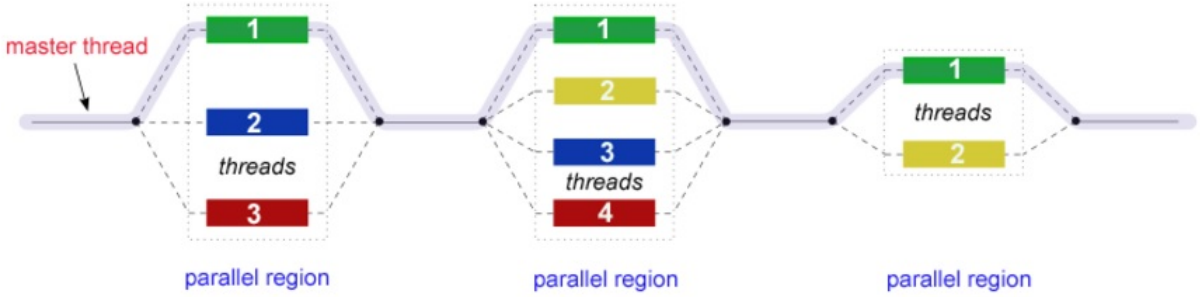
```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void mp_test(void)
{
    int my_rank = omp_get_thread num();
    int thread_count = omp_get_num_threads();
    printf("Hello from thread %d of %d\n", my_rank, thread_count);
}

int main(int argc, char *argv[])
{
    int thread count = strtol(argv[1], NULL, 10);
    //parallel block start declaration
    # pragma omp parallel num_threads(thread count)
    {
        mp_test();
    }
    //ends with closing bracket
    return 0;
}
```

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

Powyższy prosty przykład ilustruje wykorzystanie OpenMP do równoległego uruchomienia `thread_count` wątków, gdzie każdy z nich wykona funkcję `mp_test`.Dodatkowo użyta klauzula `num_threads` modyfikuję 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 zakończ wykonywanie funkcji `mp_test`. Każdy ze stworzonych wątków otrzymuje swoje id, stopień oraz parametr określający liczbę innych wątków w ramach tego samego bloku. Korzystając z funkcji `omp_get_thread_num` i `omp_num_threads` (nagłówek `<omp.h>`) otrzymywane jest id i liczba współpracujących wątków. Współdzielonym zasobem pomiędzy wątkami jest strumień wyjściowy `stdout`. [28][2]

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

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



Rysunek 3.2: Model tworzenia wątków w OpenMP[5]

Podobnie jak dla Pthreads OpenMP uwzględnia ochronę sekcji krytycznej oraz metody synchronizacji wątków. Do przeciwdziałania wyścigom i zakleszczeniom(deadlocks) pomiędzy wątkami wykorzystywane są:[8][34]

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

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

- Dyrektywa `atomic` - wykorzystuje specjalne instrukcje sprzętowe, dzięki czemu możliwa jest dużo szybsza realizacja sekcji krytycznej. Atomowe operacje to takie które wykonywane są zawsze całkowicie, bez interwencji innego wątku. Najczęściej sekcje `atomic` używane są do prostych operacji na licznikach zmienianych przez kilka wątków równolegle.[34][8]
  Operacje zmiany zmiennej:

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

  Operacje czytania i nadpisania zmiennej:

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

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

- Blokada `omp_lock_t` z pliku nagłówkowego `<omp.h>` - ogranicza dostęp do funkcji krytycznej. Posiada pięć funkcji do manipulacji blokadą [34][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. [34][28]

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

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

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

3.1.3 Wielowątkowość w standardzie C++11

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 tworzenie wieloplatformowych aplikacji wielowątkowych bez potrzeby używania dodatkowych bibliotek, takich jak Pthreads, Windows threads, OpenMP.[25][32]

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

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

Podobnie jak dla Pthreads, wielowątkowość implementowana z pomocą `std::thread`, zakłada tworzenie nowych wątków z unikalnymi id, pamięcią stosu, funkcją wykonawczą oraz listą argumentów do tej funkcji. Każdy stworzony wątek może być typu `joinable`, albo `detached`. `Joinable` oznacza, że wątek powinien zakończyć działanie przed wywołaniem swojego destruktora. Niezbędne jest zastosowanie do tego przez wątek twórcy metody `join()`, która zapewnia, że stworzony wątek wykona swoje zadanie zanim zostanie zniszczony. Jeśli pożądane jest pozwolenie wątkowi na pracę po wywołaniu jego destruktora, używana jest metoda `detach()`.[15][32]

Analogicznie do POSIX threads, do synchronizacji wykorzystywane są blokady w postaci muteksów `std::mutex` oraz zmienne warunkowe do generacji zdarzeń `std::condition_variable`. W celu usunięcia konieczności ręcznego ustawiania i odblokowywania mutexa(`mutex.lock()`, `mutex.unlock()`), wykorzystywany jest obiekt `std::lock_guard`, który po inicjalizacji muteksem jako argument, ustawia blokadę i zapewnia jej usunięcie po wyjściu z zasięgu swojej deklaracji(patrz listing 3.4).[15][18]

```
#include <thread>
#include <mutex>
#include <iostream>
#include <utility>
#include <chrono>
#include <functional>
#include <atomic>

using namespace std;

int protected_global = 0;
std::mutex protected_global_mutex;

void test_fun(int n)
{
    for (int i = 0; i < n; ++i) {
        std::cout << "Thread 1 executing\n";
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}

void safe_increment(int a, std::string &str)
{
    std::lock_guard<std::mutex> lock(protected_global_mutex);
    protected_global += a;

    cout << "Thread " << std::this_thread::get_id() << " incremented protected_global by
    " << a << '\n';
```

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

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

Innym podejściem do współbieżności, które zostało zawarte w nowym standardzie, jest współbieżność zadaniowa. Daje ona możliwość wykonania zadania - funkcji i następnie zwraca jeden wynik. Wsparcie do tego modelu jest zaimplementowane w postaci[13][25] (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`

```
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  }
```

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

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. [25][32]

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.

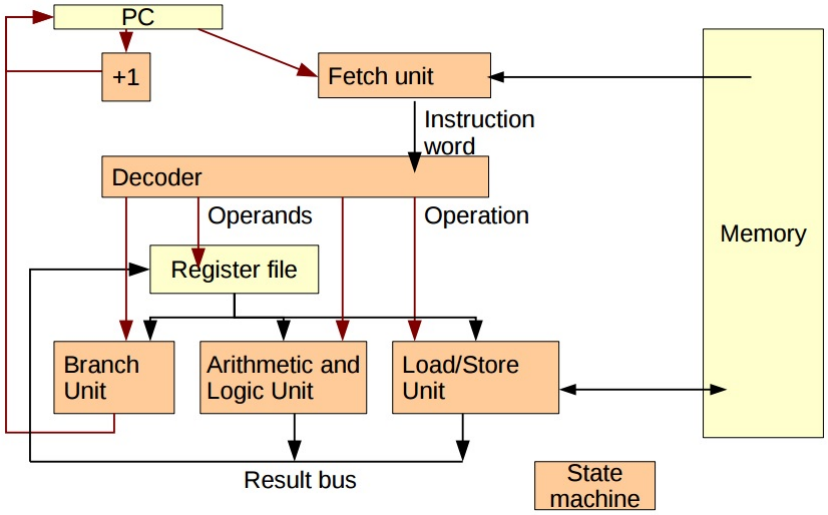
3.2 Programowanie równoległe z wykorzystaniem GPU

3.2.1 Architektura 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.[30] [21]

Każdy procesor składa się z :

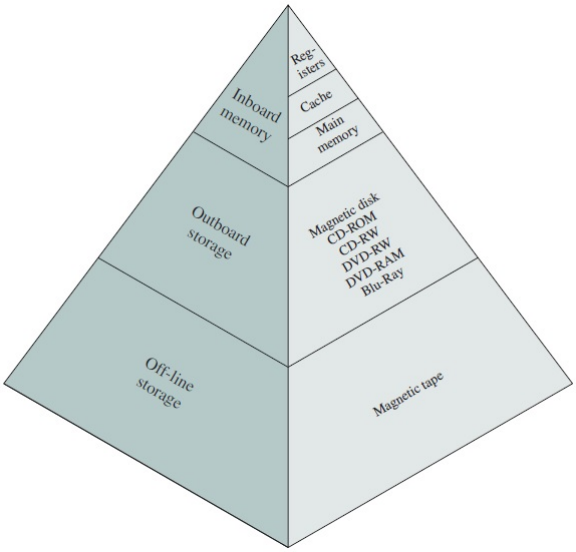
- jednostki artmetyczno logicznej - ALU(arithmetic logical unit).
- zestawu rejestrów
- jednostki kontrolnej - CU(control unit)



Rysunek 3.3: Schemat cyklu pobierania i wykonania instrukcji przez CPU [30]

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**). [30][31]

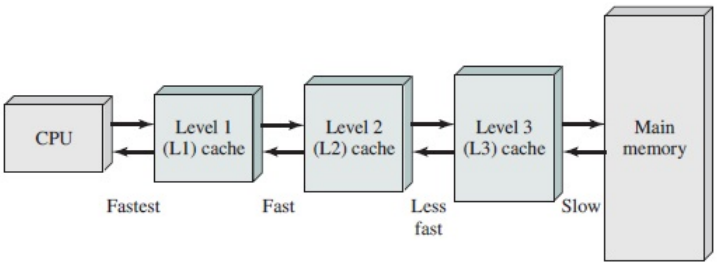
Szybkość wykonywania operacji jest w obecnych procesorach bezpośrednio powiązana z szybkością odczytu i zapisu danych pamięci komputera. Głównym problemem doboru rodzaju pamięci do zastosowania jest jej cena w zależności od pojemności i szybkości dostępu. Im większa pojemność tym wolniejsza i tańsza 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.[21][30]Dlatego mniejsza, droższa i szybsza pamięć jest uzupełniana przez większe i tańsze.



Rysunek 3.4: Hierarchia pamięci[30]

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. 3.5). 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 kopi 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. [21][30][31]



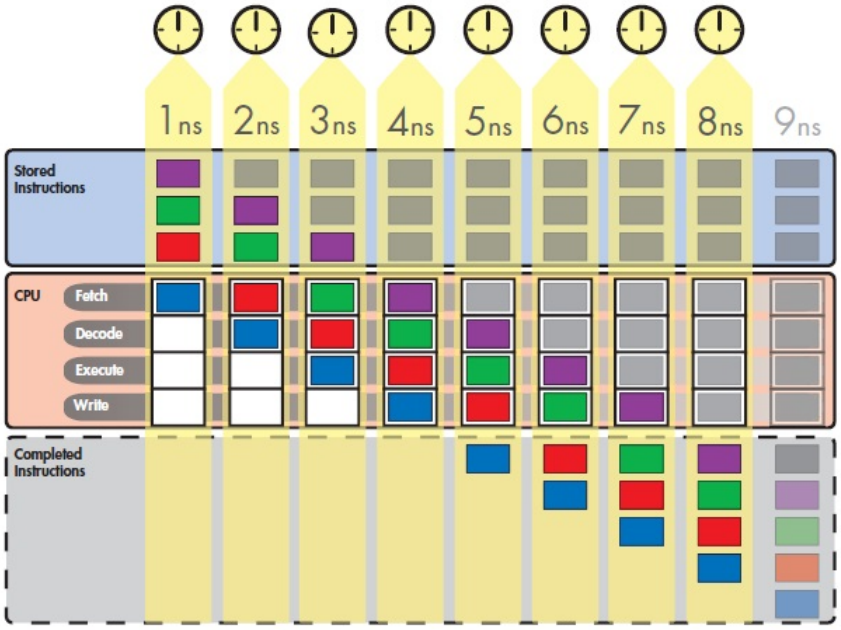
Rysunek 3.5: Trójpoziomowa organizacja pamięci cache [30]

Podstawową metodą zwiększania szybkości wykonywania instrukcji przez procesor jest wykorzystywanie 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.[31]. 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

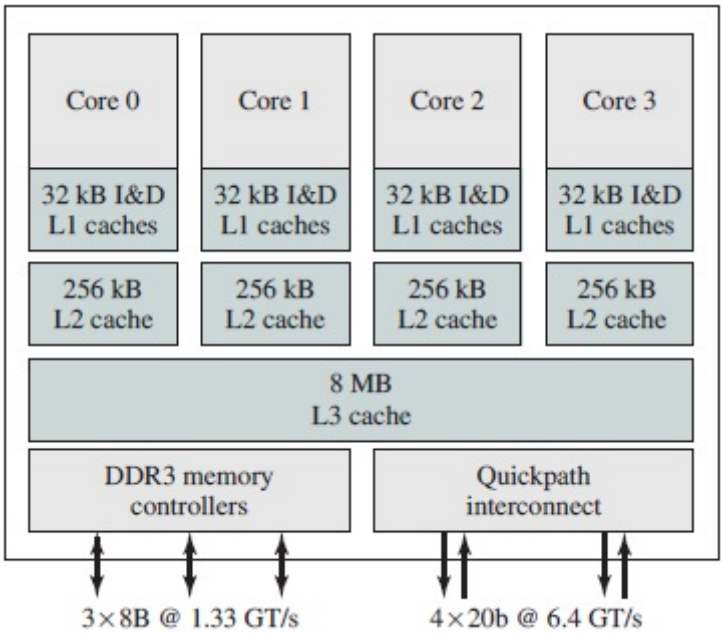
Gdy pierwsza zostaje zakończona, zaczyna się druga, a pierwsza zaczyna pobierać następną instrukcję. Analogicznie po dekodowaniu pierwszej instrukcji jest ona wykonywana w 3-ciej fazie, a w tym samym czasie druga instrukcja zaczyna być dekodowana. Zgodnie z tym tokiem postępowania w czasie pierwszego 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 dokończone oraz kolejne 3 zostaną częściowo przetworzone, a pierwsza z nowych instrukcji przejdzie przez wszystkie 4 fazy(patrz rys.3.6). [31][21]



Rysunek 3.6: Przykład 4-fazowego potoku(pipeline) procesora [31]

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 osiągów procesora z wykorzystaniem potokowości. Dalsza współbieżność wykonywania instrukcji programu została oparta na wykorzystaniu technologii wielordzeniowych.[31][21]

Innym podejściem do paralelizmu jest sprzętowa implementacja wielowątkowości - **TLP**(Thread-level parallelism). Występują trzy sprzętowe podejścia do realizacji wielowątkowości[21] (patrz rys.3.8):



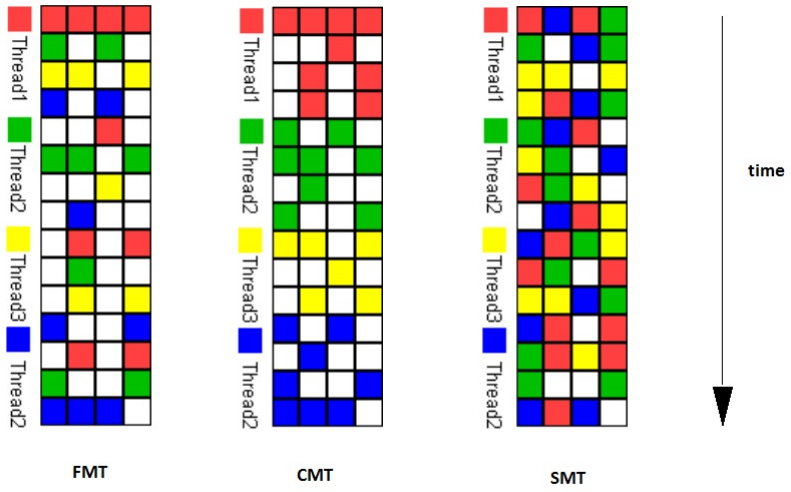
Rysunek 3.7: Procesora Intel Core i7, wykorzystujący **SMT** [30]

- 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 mutlithreading(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.
- Simultaneus mutlithreading(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.

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 jednoczesne wykonywanie wielu instrukcji, operując na wielu strumieniach danych. Składają się z kolekcji niezależnych procesorów lub rdzeni, gdzie każdy posiada własny zestaw



Rysunek 3.8: Rodzaje implementacji **TLP**

rejestrów, ALU i jednostkę kontrolną. Systemy te nie posiadają jednego zegara synchronizującego wszystkie procesory, każdy z nich może pracować asynchronicznie.[28][30]. Występują dwa rodzaje systemów MIMD: współdzielące pamięć(*shared-memory systems* oraz z oddzielną pamięcią(*distributed-memory systems*)(patrz rys.3.9). 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.[30]

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. [21][28]

3.2.2 Architektura GPU

3.2.3 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

Listings

Rozdział 9

Bibliografia

[1] Sachin B. Bhosale Amol N. Dumbare, Kiran P.Somase. Mobile robot for object detection using image processing. *International Journal of Advane Research in Computer Science and Managment Studies*, 1(6):81–84, 2013.

[2] Dieter an Mey. Parallel programming in openmp introduction. `http://scc.ustc.edu.cn/zlsc/cxyy/200910/W020100308601022991415.pdf`.

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

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

[5] Blaise Barney. Openmp. `https://computing.llnl.gov/tutorials/openMP`.

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

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

[8] OpenMP Architecture Review Board. Openmp application program interface. `http://www.openmp.org/wp-content/uploads/spec25.pdf`, 2005.

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

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

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

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

[13] C++ Concurrency. C++ reference documentation. `http://en.cppreference.com/`.

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

[15] Standard C++ Foundation. C++11 standard library extensions - concurrency. `https://isocpp.org/wiki/faq/cpp11-library-concurrency`.

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

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

[18] K Hong. Multi-threaded programming: C++11. `http://www.bogotobogo.com/cplusplus/multithreaded4_cplusplus11.php`.

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

[20] Intel. Intel processors and chipsets for embedded applications. `http://www.intel.pl/content/www/pl/pl/intelligent-systems/embedded-processors-which-intel-processor-fits-your-project.html`.

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

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

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

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

[25] Scott Meyers. *Effective Modern C++*. O'Reilly, 2015. ISBN:9781491903995.

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

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

[28] Peter S. Pacheco. *An Introduction to Parallel Programming*. Elsevier, 2011. ISBN:9780123742605.

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

[30] William Stallings. *Operating Systems Internals And Design Principes*. Prentice Hall, 2012. ISBN:9780132309981.

[31] Jon Stokes. *Inside the Machine*. No Starch Press, 2007. ISBN:9781593271046.

[32] Bjarne Stroustrup. *Język C++.Kompendium wiedzy*. Helion, 2013. ISBN:9788324685301.

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

[34] Joel Yliluoma. Guide into openmp: Easy multithreading programing for c++. `http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html`.