04.02.2023
Wydział Automatyki, Elektroniki i Informatyki
Informatyka Katowice

Raport Końcowy

Języki Asemblerowe

Usuwanie jednolitego tła

Jakub Nowakowski

1.Opis Aplikacji

Aplikacja która realizuje wykonuje usuwanie wybranego koloru przez użytkownika z określonym zakresem. Projekt jest realizowany z użyciem języka C# oraz dynamicznie dołączanej biblioteki w języku assembler. Główną problematyką była implementacja wielowątkowości i wykorzystanie instrukcji wektorowych, aby w optymalny sposób wykonywać usuwanie jednolitego tła z obrazów oraz przedstawienie zależności czasowych pomiędzy wspomnianymi językami.

2.Problematyka

- **Wielowątkowość**

  Implementacja wielowątkowości została zrealizowana w C# przy pomocy klasy „task”, jest to klasa pozwalająca na oddelegowanie zadań do wykonania wątkom, w taki sposób, że ich realizacja w czasie jest optymalizowana przez maszynę wirtualną środowiska .NET

- **Assembler**

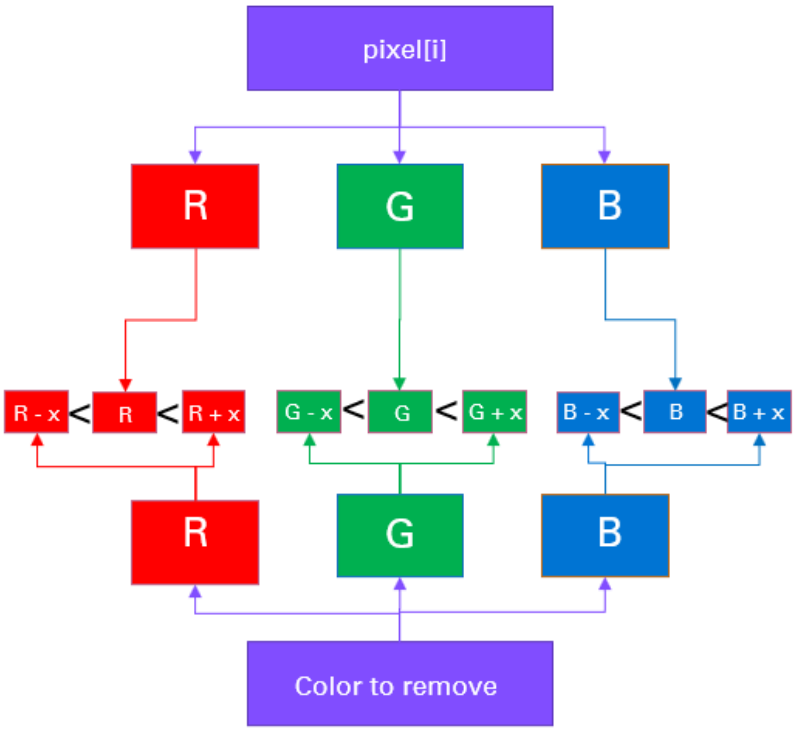
  Jednym z problemów implementacji było stworzenie działającej procedury w assemblerze która będzie mogła być wykonywana równolegle. Jednocześnie miała wykorzystywać rejestry i instrukcje wektorowe by wykorzystać pełen potencjał niskopoziomowego kodu. Kod ten musiał zostać dowiązany do języka wysokiego poziomu za pomocą dynamicznie ładowanej biblioteki.

- **Oczekiwany efekt**

  Oczekiwanym efektem była optymalizacja operacji usuwania jednolitego tła z obrazu oraz zbadanie zależności czasowej pomiędzy kodem wysokiego i niskiego poziomu. Badanie te odbywa się na różnej liczbie wątków , a jego wyniki można zapisać jako plik CSV i porównać. Obraz wynikowy również można zapisać w programie.

3.Sposób działania

- **Wykonanie w języku C#**



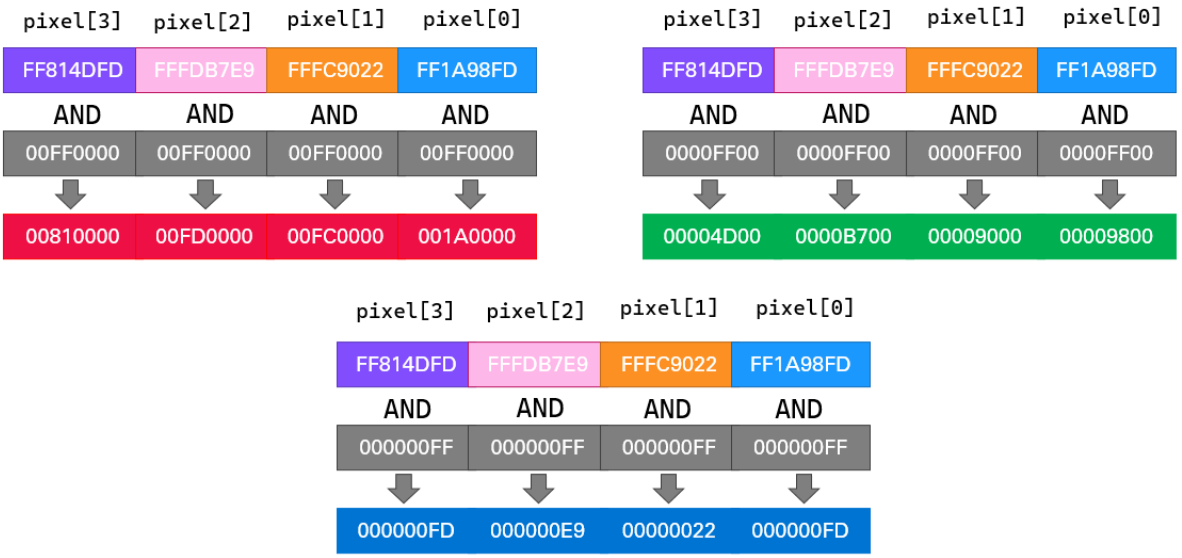
Dla części wykonywanej w języku wysokiego poziomu, dzieliliśmy każdy pixel na kanały wartości RGB , z koloru który chcemy usunąć wydzielaliśmy tak samo te kanały, jednak dodając lub odejmując margines, który został wczesnej określony. Następnie dla każdego kolejnego pixela sprawdzane było czy jego kanały mieszczą się w marginesach. Jeżeli kolor się mieścił, zastępowany był pixelem transparentnym.

Co reprezentuje poniższy kod:

```
private void doTask(int start, int end)
        {
            for (int i = start; i < end; i++)
            {

                int alpha = (pixels[i] >> 24) & 0xff;
                int red = (pixels[i] >> 16) & 0xff;
                int green = (pixels[i] >> 8) & 0xff;
                int blue = pixels[i] & 0xff;


                if ((green > _colorToRemove.G - Offset && green < _colorToRemove.G + Offset) &&
                    (red > _colorToRemove.R - Offset && red < _colorToRemove.R + Offset) &&
                    (blue > _colorToRemove.B - Offset && blue < _colorToRemove.B + Offset))
                {
                    pixels[i] = 0x00000000;
                }

            }
        }
```

- **Wykonanie w języku Assembler**

Podejście do problemu różniło się od tego realizowanego w języku wysokiego poziomu. Aby wykorzystać potencjał assemblera 64 bitowego skorzystano z instrukcji wektorowych które przetwarzały 4 pixele na raz.

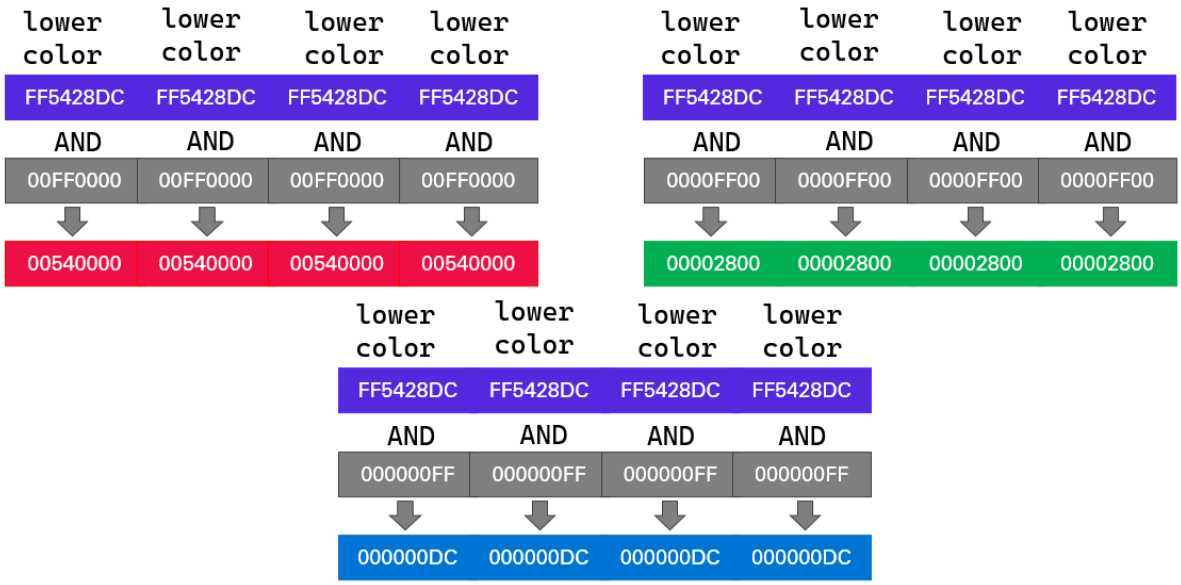


Obrazek numer 1.

Z każdego z pixeli wyekstrahowano za pomocą masek oraz operacji wektorowej AND poszczególne wartości RGB. Do xmm1 i xmm2 przekazane zostały rejestrami 32 bitowymi wartości górnej i dolnej granicy koloru. Następnie wypełniono te rejestry , aby każde 32 bity były tymi kolorami, co widoczne jest na poniższym obrazku.

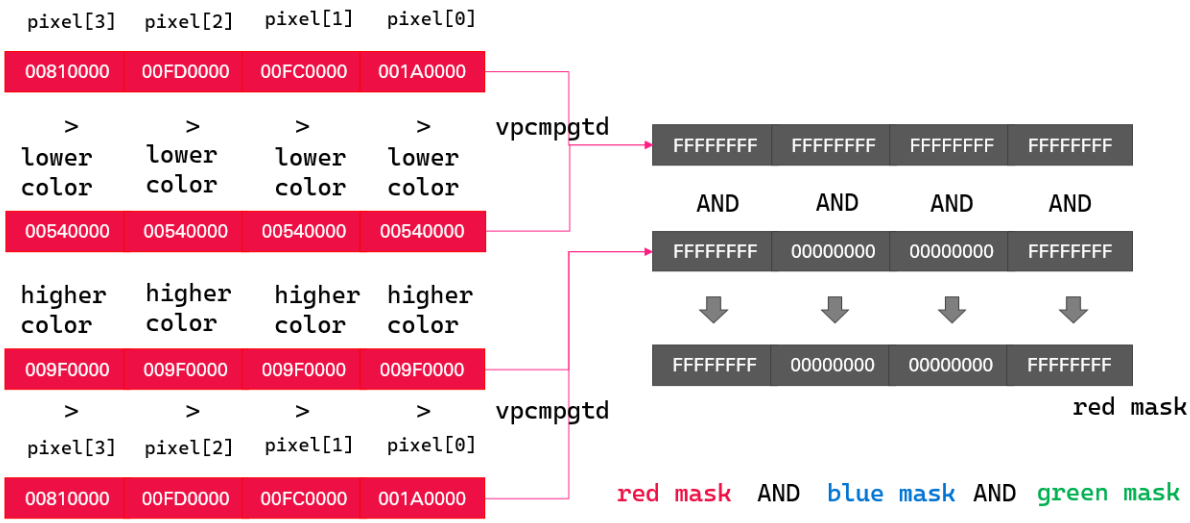
Obrazek numer 2.

Dla każdego koloru granicznego wykonano operację identyczną jak na obrazku numer 1. Dla wszystkich kanałów RGB wyekstrahowano poszczególne wartości za pomocą maski i operacji AND, przykład takiego wyniku dla koloru niższej granicy zaprezentowano na obrazku numer 3.



Obrazek numer 3.

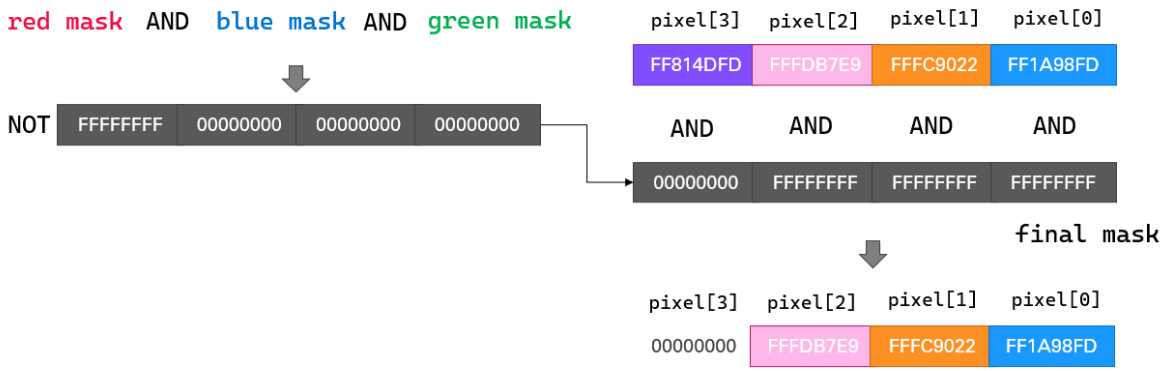
Następnie dla każdego z wydzielonych kanałów wykonywane było porównanie za pomocą instrukcji **vpcmpgtd** instrukcja ta porównywała 32 bitowe liczby całkowite ze znakiem i zawracała wynik w postaci 0xFFFFFFFFh gdy odpowiadające wartości są większe lub 0x00000000h gdy wartości są mniejsze. Problem mógł się pojawić gdyby jakiś kolor należał do wartości ujemnej liczby całkowitej. Problemu udało się uniknąć dzięki zastosowaniu formatu koloru ARGB gdzie przy zamaskowaniu kanału alpha w raz z innymi kolorami nie wkraczaliśmy na liczby ujemne.



Obrazek numer 4.

W wyniku przedstawionych na obrazku numer 4 porównań dla przykładowego kanału czerwonego otrzymywaliśmy dwie maski, aby otrzymać finalną maskę która będzie mówić o tym czy dany kanał znajduje się w zakresie, należało wykonać wektorową operację AND na obu tych maskach.

Gdy posiadaliśmy maski już dla wszystkich kanałów należało wykonać na nich operacje AND wektorową. Otrzymana maska niestety posiadała 0xFFFFFFFF tam gdzie wszystkie warunki zostały spełnione. Aby zamienić pixele które spełniają wszystkie warunki na transparentne należało wykonać operacje negacji na tej masce.



Obrazek numer 5.

Ostatecznie w wyniku wektorowej operacji AND finalnej maski i wczytanych pixeli otrzymywaliśmy odpowiedni wynik gdzie pixel który był w zakresie został transparenty.

Omówione działanie wykonane jest w kodzie:

```
.data
mask_for_red dd 000FF0000h
mask_for_green dd 00000FF00h
mask_for_blue dd 0000000FFh

.code
remove_color_asm proc

    movdqu xmm0, [RCX]          ;moving 4 pixels from array to xmm0

    movd xmm1, edx              ;moving 32bit color of low boundry to xmm1
    pshufd xmm1, xmm1, 0        ;fill in all places in xmm1 with this pixel

    movd xmm2, r8d              ;moving 32bit color of high boundry to xmm2
    pshufd xmm2, xmm2, 0        ;fill in all places in xmm2 with this pixel

;---------------------- red section ----------------------------------------


    movd xmm11, [mask_for_red]          ; copying 32bit mask that lefts red to xmm11
    pshufd xmm11, xmm11, 0              ; fill in all places in xmm11 with this mask

    vpand xmm3, xmm1, xmm11             ; leaving red channel of low boundry and save to xmm3
    vpand xmm6, xmm2, xmm11             ; leaving red channel of high boundry and save to xmm6
    vpand xmm15, xmm0, xmm11            ; leaving red channel of pixels and save to xmm15

    vpcmpgtd xmm3, xmm15, xmm3          ; check if red color of pixels is above red color of low boundry
    vpcmpgtd xmm6, xmm6, xmm15          ; check if red color of high boundry is above red color of pixels

    vpand xmm7, xmm3, xmm6              ; saving resutl mask fater AND operation


;-------------------------------------------------------------------------(result mask saved in xmm7)


;---------------------- green section ---------------------------------------


    movd xmm12, [mask_for_green]        ; copying 32bit mask that lefts green to xmm12
    pshufd xmm12, xmm12, 0              ; fill in all places in xmm12 with this mask

    vpand xmm3, xmm1, xmm12             ; leaving green channel of low boundry and save to xmm3
    vpand xmm6, xmm2, xmm12             ; leaving green channel of high boundry and save to xmm6
    vpand xmm15, xmm0, xmm12            ; leaving green channel of pixels and save to xmm15

    vpcmpgtd xmm3, xmm15, xmm3          ; check if green color of pixels is above green color of low boundry
    vpcmpgtd xmm6, xmm6, xmm15          ; check if green color of high boundry is above green color of pixels

    vpand xmm8, xmm3, xmm6              ; saving resutl mask fater AND operation
```

```
;-------------------------------------------------------------------------(result mask saved in xmm8)


;---------------------- blue section ----------------------------------------


    movd xmm13, [mask_for_blue]          ; copying 32bit mask that lefts blue to xmm13
    pshufd xmm13, xmm13, 0               ; fill in all places in xmm13 with this mask

    vpand xmm3, xmm1, xmm13              ; leaving blue channel of low boundry and save to xmm3
    vpand xmm6, xmm2, xmm13              ; leaving blue channel of high boundry and save to xmm6
    vpand xmm15, xmm0, xmm13             ; leaving blue channel of pixels and save to xmm15

    vpcmpgtd xmm3, xmm15, xmm3           ; check if blue color of pixels is above blue color of low bound
    vpcmpgtd xmm6, xmm6, xmm15           ; check if blue color of high boundry is above blue color of pixels

    vpand xmm9, xmm3, xmm6               ; saving resutl mask fater AND operation

;-------------------------------------------------------------------------(result mask saved in xmm9)



;---------------------- summary section ----------------------------------------

    vpand xmm3, xmm8, xmm7               ; AND operation on masks
    vpand xmm6, xmm9, xmm3               ; AND operation on masks and saving final to xmm6

;-------------------------------------------------------------------------(final mask in xmm6)


    vpcmpeqd xmm5, xmm5, xmm5   ; negation of the mask
    pandn xmm6, xmm5

    vpand xmm0, xmm0, xmm6      ; AND operation on pixels (xmm0) and mask (xmm4) with result saved in xmm0
    movdqu [RCX], xmm0          ; modified 4 pixels saved back to array

    ret

remove_color_asm endp
end
```

- **Realizacja wielowątkowości**

Dla kodu w assemblerze oraz w C# wielowątkowości została zrealizowana w praktycznie identyczny sposób, w związku z tym omówiona zostanie na przykładzie gdzie wykorzystywany jest assembler, który został wybrany w związku z większym stopniem złożoności.

```
ThreadPool.SetMaxThreads(numThreads, numThreads);
ThreadPool.SetMinThreads(numThreads, numThreads);
_splitSize = numThreads;

int bitmapSize = bitmap.Width * bitmap.Height;

int sizeOfPixelsArray = (((bitmapSize) / 4) * 4)+4;

int chunkSize = ((sizeOfPixelsArray / _splitSize) / 4) * 4;

pixels = new int[sizeOfPixelsArray];

_colorToRemove = colorPassed;

lowColorBoundry = (255 << 24) | (Math.Max(0,_colorToRemove.R - offset) << 16) |
(Math.Max(0, _colorToRemove.G - offset) << 8) | (Math.Max(0, _colorToRemove.B -
offset));

highColorBoundry = (255 << 24) | (Math.Min(255, _colorToRemove.R + offset) <<
16) | (Math.Min(255, _colorToRemove.G + offset) << 8) | (Math.Min(255,
_colorToRemove.B + offset));
```

Na początku należało ograniczyć ilość dostępnych wątków dla „tasków” , a następnie liczbę fragmentów na które będzie podzielony obraz określono jako liczbę wątków. Zapisano wielkość bitmapy oraz wyliczono wielkość tablicy pixeli jako najbliższą większą liczbę podzielną przez 4. Określono części na jakie fragmenty bitmapa będzie podzielona. Utworzono tablice pixeli. Zapisano przekazany kolor do usunięcia. Utworzono dwa kolory graniczne , dla jednego z odjętym marginesem , a dla drugiego z dodanym dla każdego kanału.

```
var tasks = new Task[_splitSize+1];

DateTime beggingDateTime = DateTime.Now;

for (int i = 0; i < _splitSize; i++)
   {
     int start = i * chunkSize;
     int end = (i + 1) * chunkSize;

     tasks[i] = Task.Factory.StartNew(() => doTask(start,chunkSize/4));

    }
int donePixels = _splitSize * chunkSize;

 if (_splitSize * chunkSize < pixels.Length)
  {
    tasks[_splitSize] = Task.Factory.StartNew(() => doTask(donePixels - 1,
(pixels.Length - donePixels) /4));
  }
  else
  {
     tasks[_splitSize] = Task.Factory.StartNew(() => { return; });
  }

Task.WaitAll(tasks);

DateTime endDateTime = DateTime.Now;

TimeElapsed = (endDateTime - beggingDateTime);
```

W tym fragmencie była tworzona tablica „tasków” o jeden większa niż liczba na która dzieliliśmy bitmapę. Każdemu „taskowi” przydzielany następnie fragment tablicy na którym miał operować. Gdy został fragment tablicy który nie był zrealizowany wykorzystywany do tego był dodatkowy „task”, w przeciwnym wypadku ten dodatkowy „task” nie realizował nic. Następnym krokiem było poczekanie na wszystkie „taski” i zakończenie pomiaru czasu.

Podjęta była próba przydzielenia jednemu „taskowi” wykonania jednej procedury asemblerowej jednak narzut czasowy stworzenia i oddelegowania „tasku” okazał się znacznie większy niż czas wykonania procedury. Optymalniejszym więc podejściem okazało się wykonanie w pętli w „tasku” całej wielkości fragmentu bitmapy.

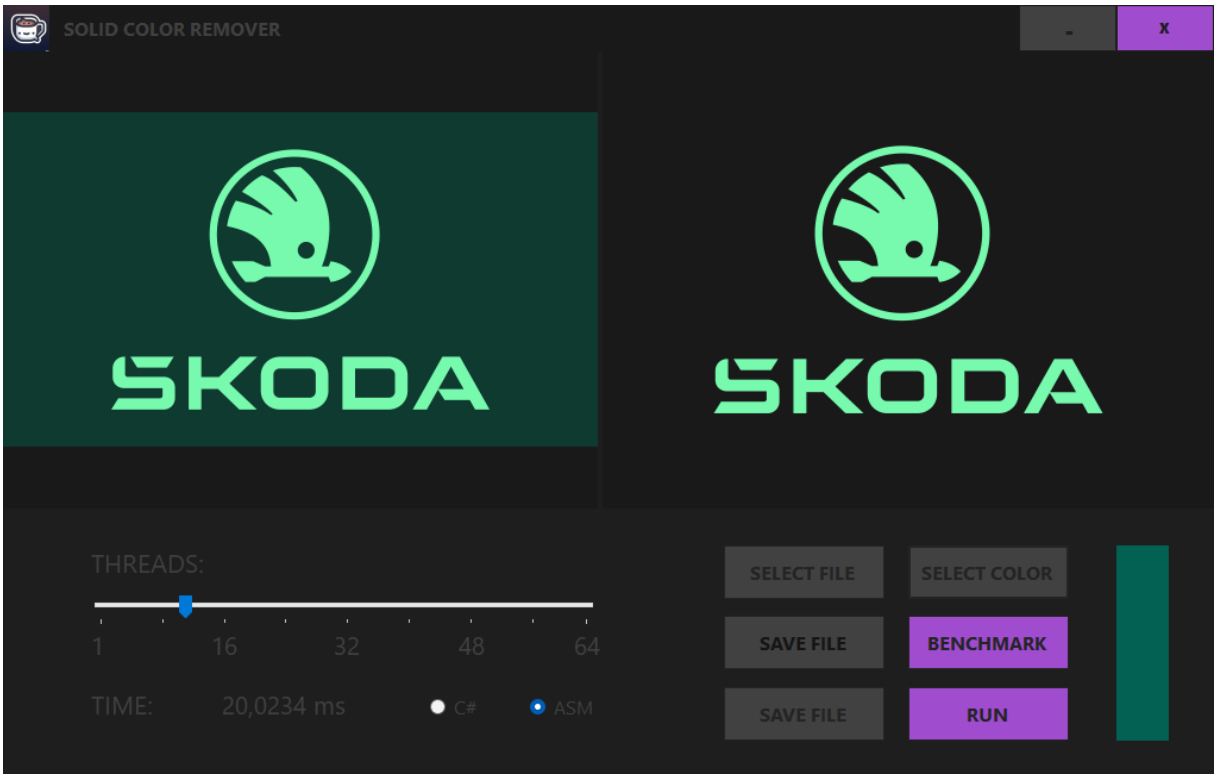
Tak prezentowała się funkcja wykonywana w jednym tasku:

```
private void doTask(int start, int end)
        {
            fixed (int* ptr = &pixels[start])
            {
               for (int i = 0; i < end; i++)
                {
                  RemoveColor(ptr+(i*4), lowColorBoundry, highColorBoundry);
                }
             }
        }
```

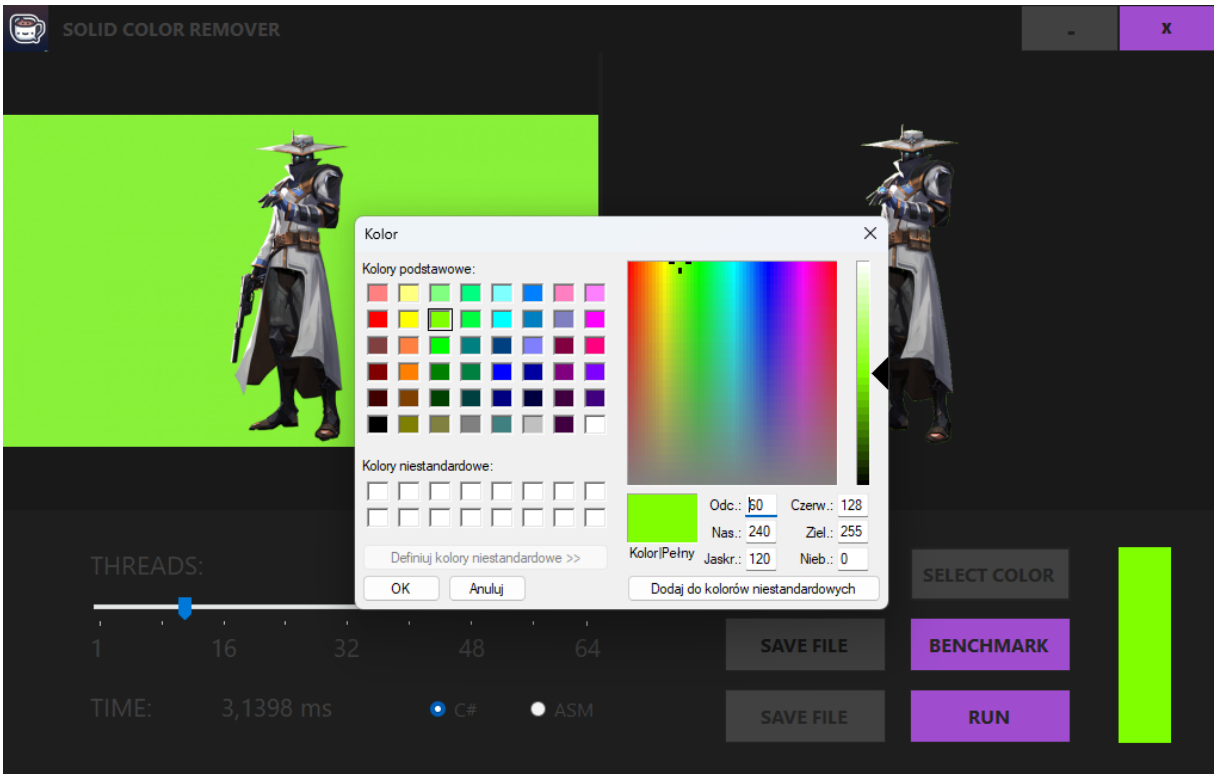
Przed operacją na tablicy należało umieścić w kodzie fragment „fixed”. Gwarantował on, że tablica pod który odwoływała się procedura asemblerowa poprzez wskaźnik, pozostanie w tym samym miejscu w pamięci przez cały fragment objętym słowem kluczowym fixed.

4.Graficzny interfejs użytkownika

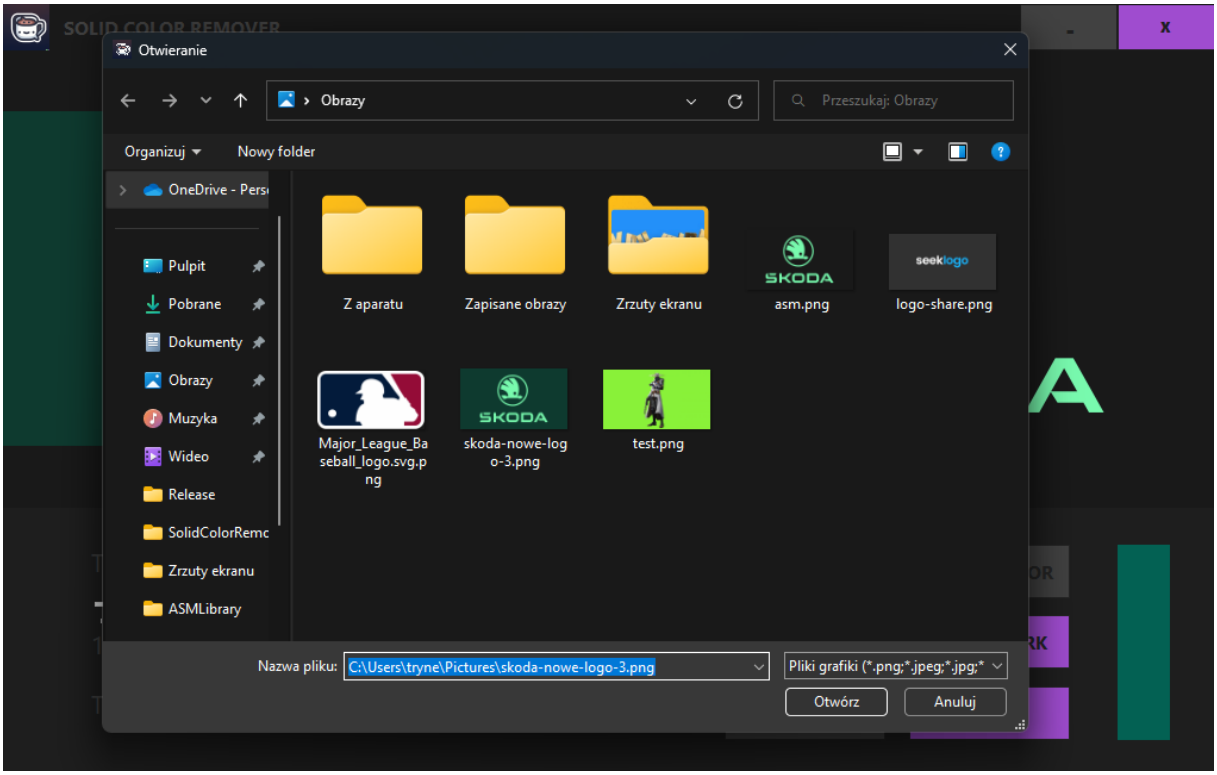
- **Główny widok po włączeniu aplikacji**



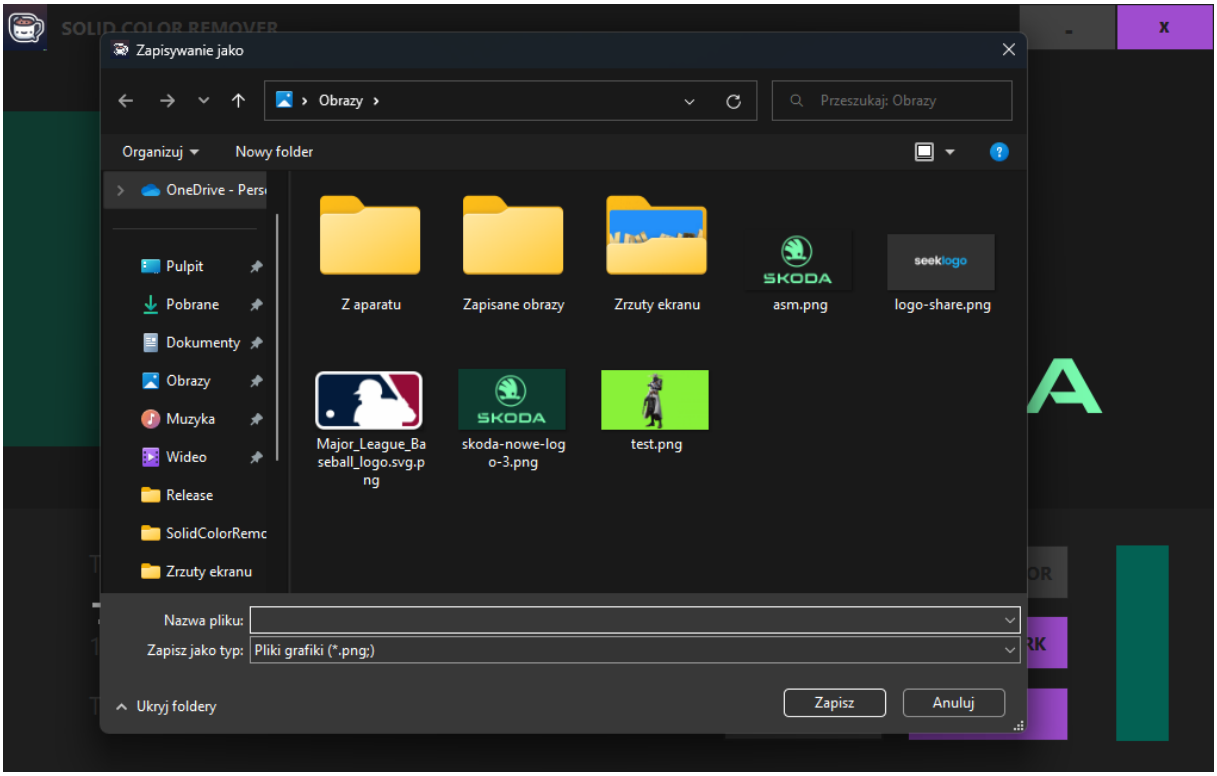
- **Okno wyboru koloru**



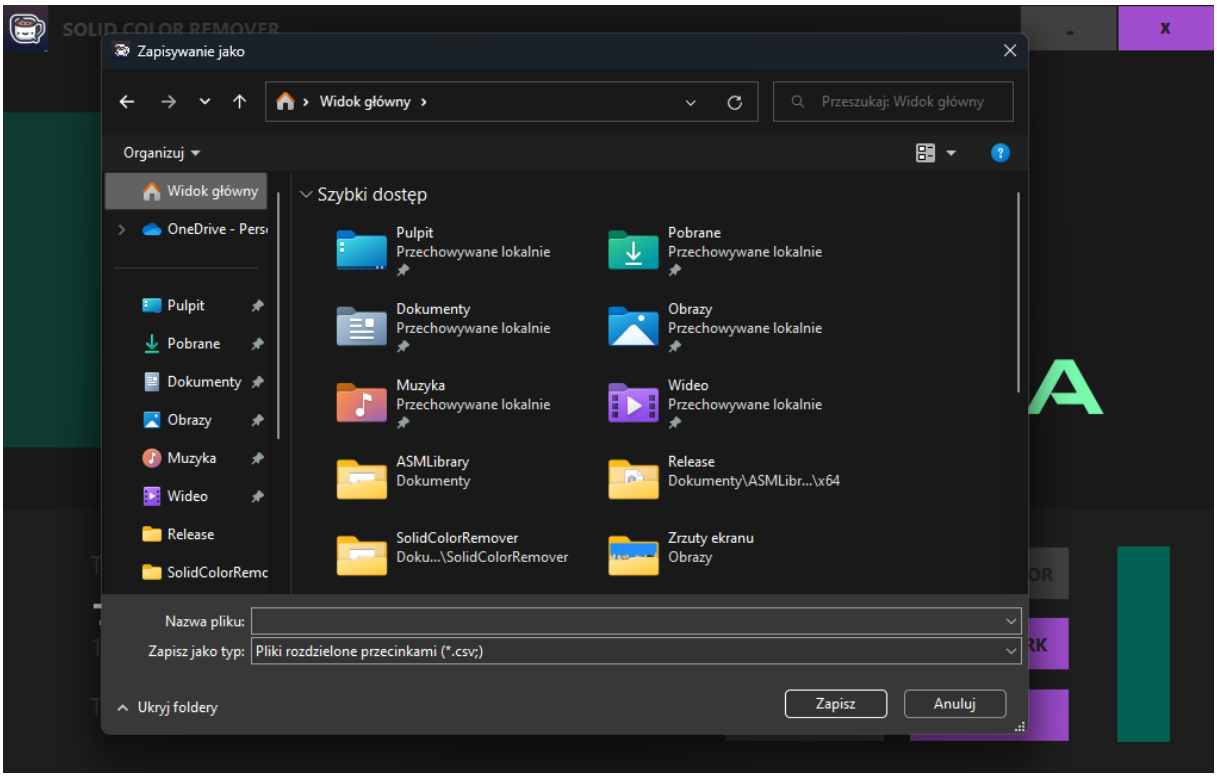
- **Okno wyboru zdjęcia**



- **Okno zapisu zdjęcia wynikowego**

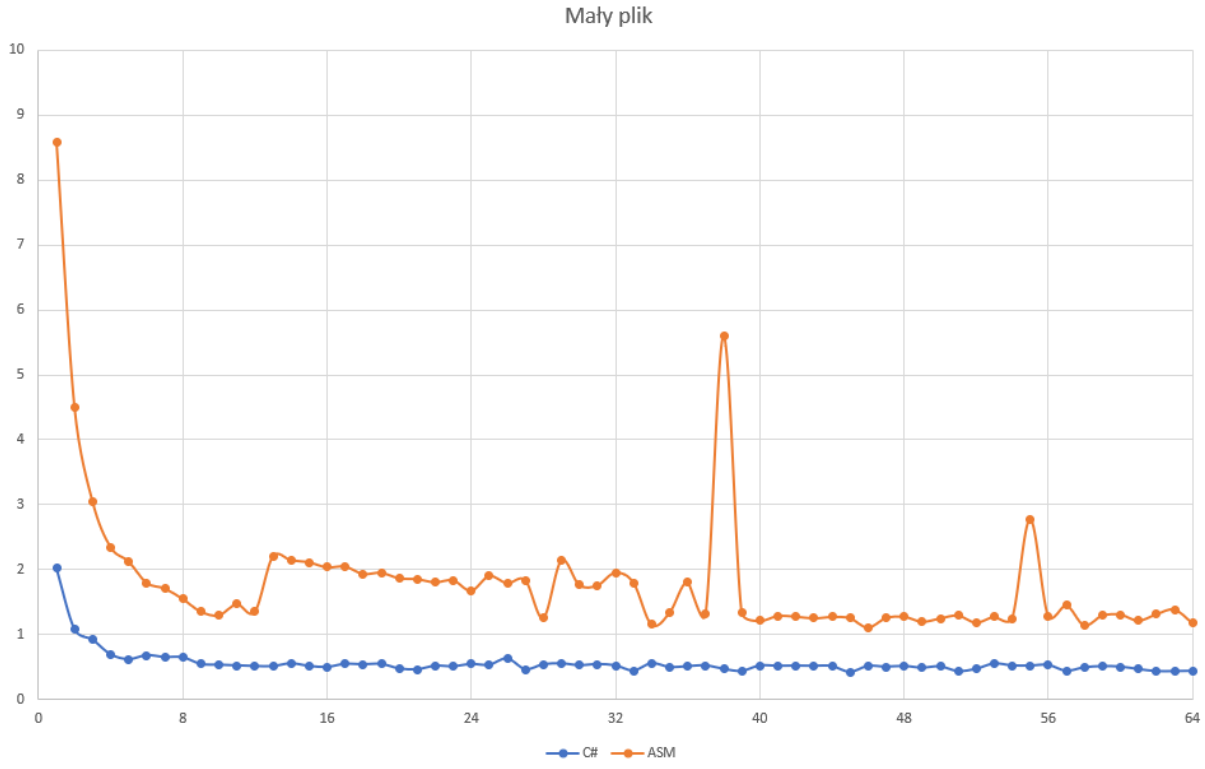


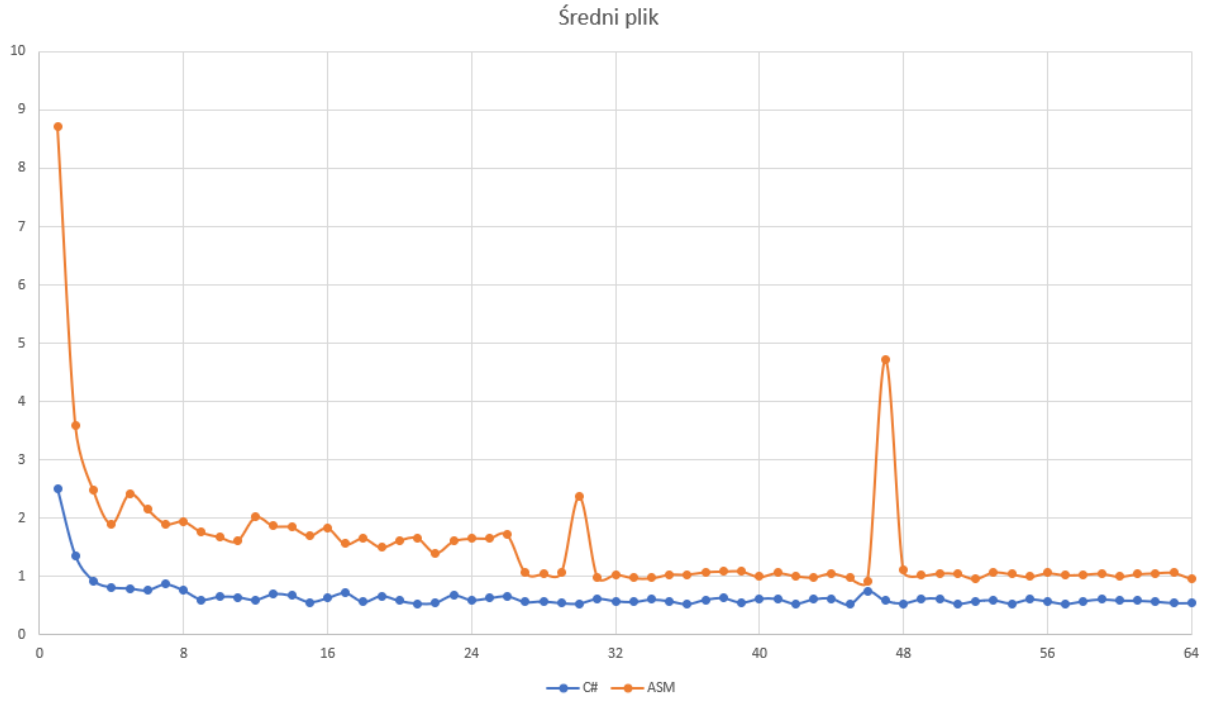
- **Okno zapisu wyników testów w postaci pliku CSV**

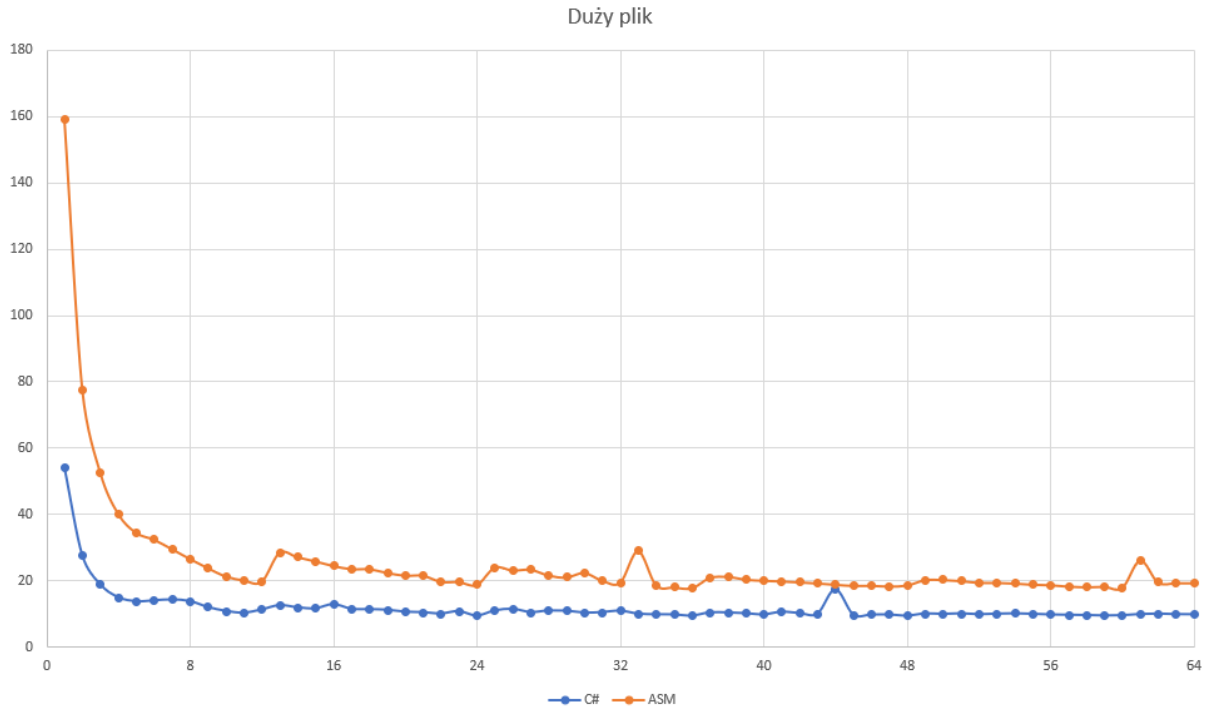


5.Testy

W ramach testów przeprowadzono 5 uśrednionych pomiarów dla każdej ilości wątków. Pomiary te wykonano dla 3 różnych rozmiarów plików. Plik najmniejszy był wielkości 28KB, średni 283KB, a największy 393KB. Zależności czasowe zostały przedstawione na poniższych wykresach. Całość testów została wykonana na procesorze Ryzen 5 3600.







Czas wykonania w języku asemblerowym był około 1,5-2 razy dłuższy co widoczne jest na wykresach. Zauważalny jest spadek czasu wykonania w przypadku gdy liczba wątków w programie odpowiadała liczbie procesorów logicznych. Było to zgodne z oczekiwaniami ponieważ w takim przypadku najlepiej wykorzystujemy zasoby procesora. Możemy również wywnioskować że zależność czasowa od rozmiaru pliku jest nie liniowa – czas wzrasta szybciej niż liniowo.

6.Wnioski

Głównym wnioskiem wynikającym z projektu jest to jak dobrze działają optymalizacja dzisiejszych kompilatorów. Osiągnięcie czasów bliskich czasom wykonywania kodu wysokiego poziomu, było bardzo trudne oraz wymagało dobrego zaznajomienia się z wektorowymi instrukcjami asemblera 64 bitowego. Miłym zaskoczeniem była wygoda realizacji wielowątkowości dzięki narzędziom dostarczonym przez platformę .NET. Wykonanie aplikacji łączącej te dwa języki pokazało, gdzie zastosowanie tak niskopoziomowego języka jakim jest assembler, może mieć swoją niszę. Po dokładnym zaplanowaniu kodu nie był potrzebne wykorzystanie debugera ponieważ kod był przewidywalny i wykonywał dokładnie to, co było w nim zapisane. Pozwala to na eliminację nieprzewidzianego zachowania oraz błędów wynikających z korzystania z dostarczonych bibliotek. Gdzie w systemach prostych i wykonujących krytyczne operacje może okazać się bardzo dużą zaletą. W porównaniu do języków wysokiego poziomu, był dużo mniejszy dostęp do materiałów związanych z realizacją kodu , szczególnie dla assemblera 64 bitowego. Wykonanie finalnej procedury wiązało się z dużą ilością czasu poświęconego na szukanie informacji lub zrozumienie dokumentacji.