Instytut Informatyki Politechniki Śląskiej
Zespół Mikroinformatyki i Teorii Automatów Cyfrowych

Laboratorium SMiW

Rok akademicki	Rodzaj studiów*: SSI/NSI/NSM	Numer ćwiczenia:	Grupa	Sekcja
2019/2020	SSI	JA	3	1

Imię	Filip	Prowadzący:	KH
Nazwisko	Plachta		

Raport końcowy

Temat projektu:

Zmiana jasności obrazu

Data oddania: dd/mm/rrrr	**05/02/2020**

1. Opis projektu

Celem projektu było zrealizowanie programu, zmieniającego jasność obrazu.
Podstawowe założenia określone na początku:

- możliwość zmiany jasności
- możliwość wyboru ilości wątków (1-64)
- wykrywanie liczby wątków
- GUI napisane w języku C#
- Rozjaśnianie wykonywane w dwóch bibliotekach:
  - C++
  - ASM
- użycie instrukcji wektorowych w powyższych bibliotekach
- możliwość zapisania nowego obrazu do nowego pliku

2. Analiza wykonania projektu

Rozjaśnianie obrazu jest to najprościej mówiąc powiększanie wartość RGB obrazu. Może być to zrealizowane zarówno używając pewnego czynnika i mnożąc lub dodając go do każdej wartości RGB. W tym projekcie zdecydowałem się na drugą opcję względu na to, dodawanie jest prostszą i szybszą operacją.

Każda wartość RGB jest liczbą naturalną z zakresu [0 ; 255]. Aby umożliwić całkowite rozjaśnienie oraz ściemnienie, czynnik powinien przyjąć wartość z zakresu [-255 ; 255].

Dodawanie powinno odbywać się z tzw. saturacją, czyli nie powinno przekraczać wartości 0 i 255. Każda wartość RGB piksela to bajt. Na 1 bajcie można zapisać liczbę z zakresu [-127;128] lub [0; 255]. A zatem program można zrealizować w dwojaki sposób:

- jedna funkcja w CPP/ASM dodająca czynnik, będący liczbą z zakresu [-127; 128]
- dwie funkcje w CPP/ASM – jedna dodająca, druga odejmująca czynnik, będący liczbą z zakresu [0; 255]

Zdecydowałem się na opcję numer 2, ponieważ umożliwia ona większy zakres zmiany jasności, pomimo większej ilości kodu.
Funkcja w ASM/CPP powinna być wywołana 3 razy – dla wartości R, G i B i powinno to dziać się w pętli aż skończy się tablica RGB. Pętla ta będzie zrównoleglona, tzn. będzie operowała na wątkach. Najlepszym sposobem będzie użycie pętli ParalellFor, ponieważ jest prosta w użyciu i od razu generuje index.

Następną kwestią są operacje wektorowe. Rejestry używane do operacji wektorowych(XMM) mają pojemność 128 bitów, a zatem 16 bajtów – 16 wartości R, G lub B. Wartości te są dodawane/odejmowane z utworzonym wektorem 16 wartości czynnika.

GUI zostało zrealizowane za pomocą WindowsForms, również ze względu na prostotę tej biblioteki.

3. Realizacja

Algorytm realizowany przez bibliotekę C++ / ASM:

- Załaduj do rejestru XMM_a 16 wartości R/G/B przekazanych w tablicy.
- Zapełnij rejestr XMM_b wartością czynnika
- Zsumuj/Odejmij rejestry **z saturacją**
- zapisz rejestr XMM_a do tablicy z której przyszły wartości R/G/B.

Funkcje w C#:

```
[DllImport("C_PROC.dll")]
3 references
static extern void LightenC(IntPtr tab, byte ratio);

[DllImport("C_PROC.dll")]
3 references
static extern void DimC(IntPtr tab, byte ratio);

[DllImport("ASM_PROC.dll")]
3 references
static extern void LightenASM(IntPtr tab, byte ratio);
[DllImport("ASM_PROC.dll")]
3 references
static extern void DimASM(IntPtr tab, byte ratio);
```

Funkcja Lighten – realizuje dodawanie, rozjaśnia.
Funkcja Dim – realizuje odejmowanie, sciemnia.
Argumenty:

- IntPtr tab – wskaźnik na tablicę 16 bajtów
- byte ratio – wartość którą należy dodać/odjąć

Realizacja w C++:

```
void LightenC(unsigned char* tab, unsigned char val)
{

    __m128i* l = (__m128i*)tab;                 // załadowanie tablicy 16 pikseli do rejestru xmm l
    __m128i* r = &_mm_set1_epi8(val);           // wypełnienie  rejestru xmm r 16 wartościami val(wspolczynnik zmiany jasnosci)
    _mm_store_si128(l, _mm_adds_epu8(*l, *r));  // zsumowanie dwóch rejestrów xmm (r+l), a następnie zapisanie ich w xmm l
    return;
}

void DimC(unsigned char* tab, unsigned char val)
{

    __m128i* l = (__m128i*)tab;                 // załadowanie tablicy 16 pikseli do rejestru xmm l
    __m128i* r = &_mm_set1_epi8(val);           // wypełnienie  rejestru xmm r 16 wartościami val(wspolczynnik zmiany jasnosci)
    _mm_store_si128(l, _mm_subs_epu8(*l, *r));  // zsumowanie dwóch rejestrów xmm (r+l), a następnie zapisanie ich w xmm l
    return;
}
```

Realizacja w ASM:

```
.code
LightenASM PROC
    MOVDQU XMM0, [RCX]          ; załadowanie 16 pikseli do rejestru XMM0
    MOVQ XMM2, RDX              ; załadowanie wartości zmiany jasności do rejestru xmm 2
    MOV EAX, 0                  ; załadowanie 0 do eax (licznik petli)
    L1:                         ; petla zapelniajaca kazdy bajt xmm1 wartoscia zmiany jasnosci
        PADDB XMM1, XMM2        ; dodanie xmm2 na ostatnie 8 bitow xmm1
        PSLLDQ XMM1, 1          ; przesuniecie rejestru xmm1 bitowe o 1 bajt w lewo
        ADD EAX, 1              ; inkrementacja licznika petli
        CMP EAX, 16             ; sprawdzanie warunku koncowego petli (16 iteracji - rejestr xmm1 - 128 bitow = 16 bajtow)
        JL L1                   ; powrot do poczatka petli
    PADDB XMM1, XMM2            ; ostatnie dodanie xmm2 do xmm1
    PADDUSB XMM0, XMM1          ; zsumowanie rejestru zawierającego piksele z rejestrem zawierającym wartości zmiany jasności
    MOVDQU [RCX], XMM0          ; załadowanie nowych wartości pikseli z powrotem do tablicy
    RET
LightenASM ENDP

DimASM PROC
    MOVDQU XMM0, [RCX]          ; załadowanie 16 pikseli do rejestru XMM0
    MOVQ XMM2, RDX              ; załadowanie wartości zmiany jasności do rejestru xmm 2
    MOV EAX, 0                  ; załadowanie 0 do eax (licznik petli)
    L1:                         ; petla zapelniajaca kazdy bajt xmm1 wartoscia zmiany jasnosci
        PADDB XMM1, XMM2        ; dodanie xmm2 na ostatnie 8 bitow xmm1
        PSLLDQ XMM1, 1          ; przesuniecie rejestru xmm1 bitowe o 1 bajt w lewo
        ADD EAX, 1              ; inkrementacja licznika petli
        CMP EAX, 16             ; sprawdzanie warunku koncowego petli (16 iteracji - rejestr xmm1 - 128 bitow = 16 bajtow)
        JL L1                   ; powrot do poczatka petli
    PADDB XMM1, XMM2            ; ostatnie dodanie xmm2 do xmm1
    PSUBUSB XMM0, XMM1          ; odejmowanie rejestru zawierającego piksele z rejestrem zawierającym wartości zmiany jasności
    MOVDQU [RCX], XMM0          ; załadowanie nowych wartości pikseli z powrotem do tablicy
    RET
DimASM ENDP
END

```

Jedyną różnicą w realizacji jest zapełnianie rejestru XMM czynnikiem. W C++ jest to realizowane za pomocą gotowej instrukcji _mm_set1_epi8, natomiast w ASM nie była dostępna gotowa instrukcja. Zrealizowałem to jednak za pomocą prostej pętli, powtórzonej 16 razy:

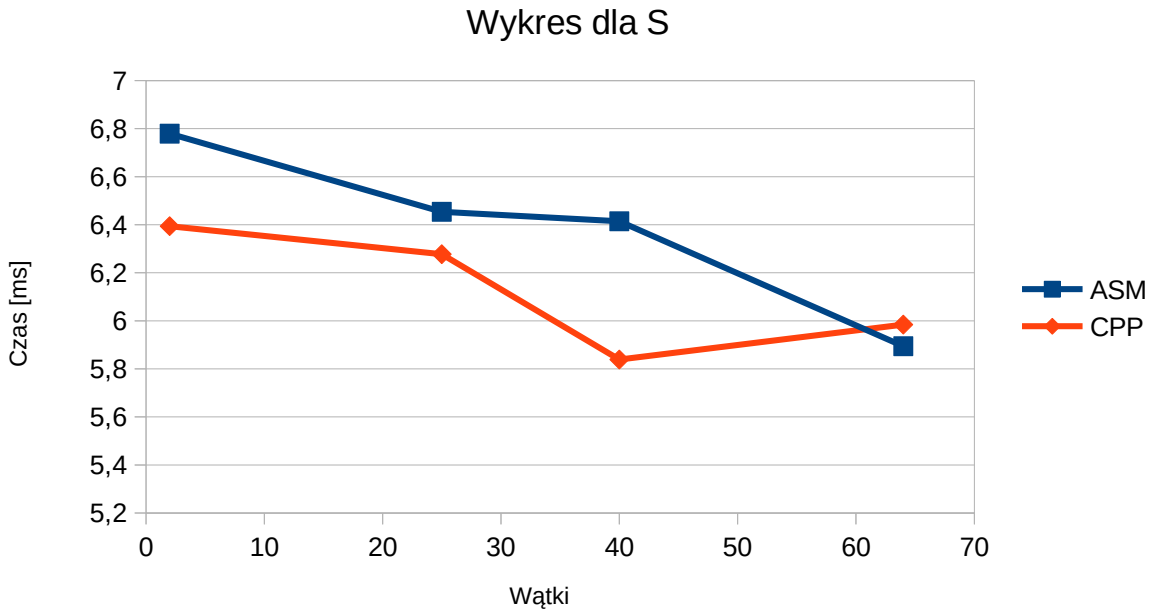
- dodaj wartość do rejestru
- przesunięcie bitowe o 8 w lewo

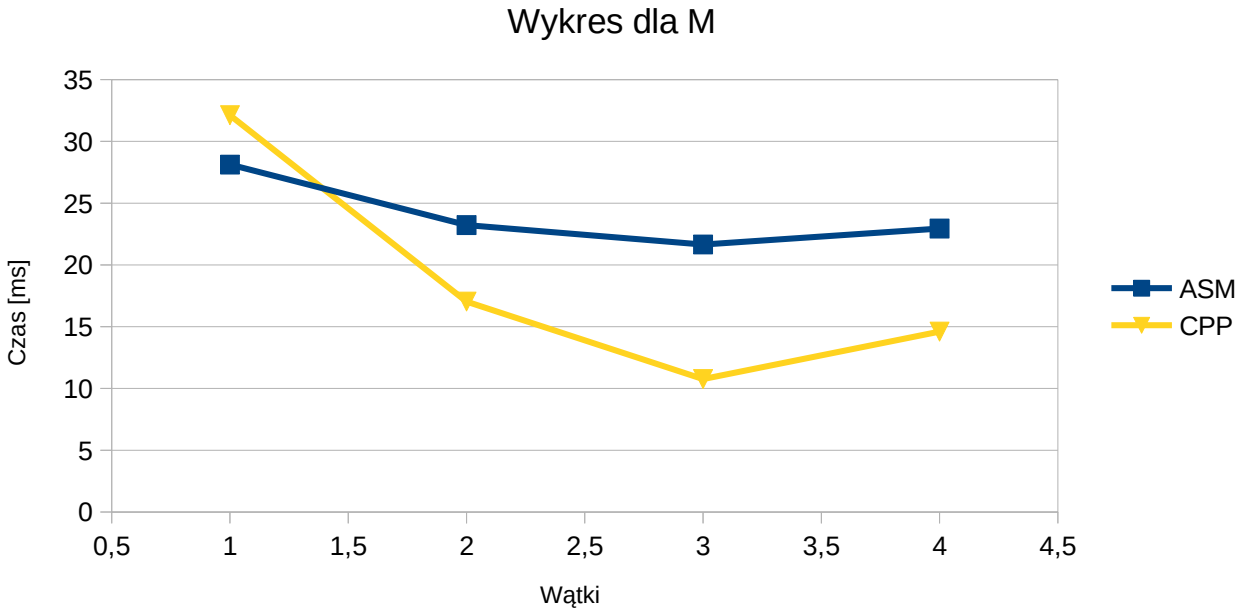
4. Analiza i porównanie czasowe

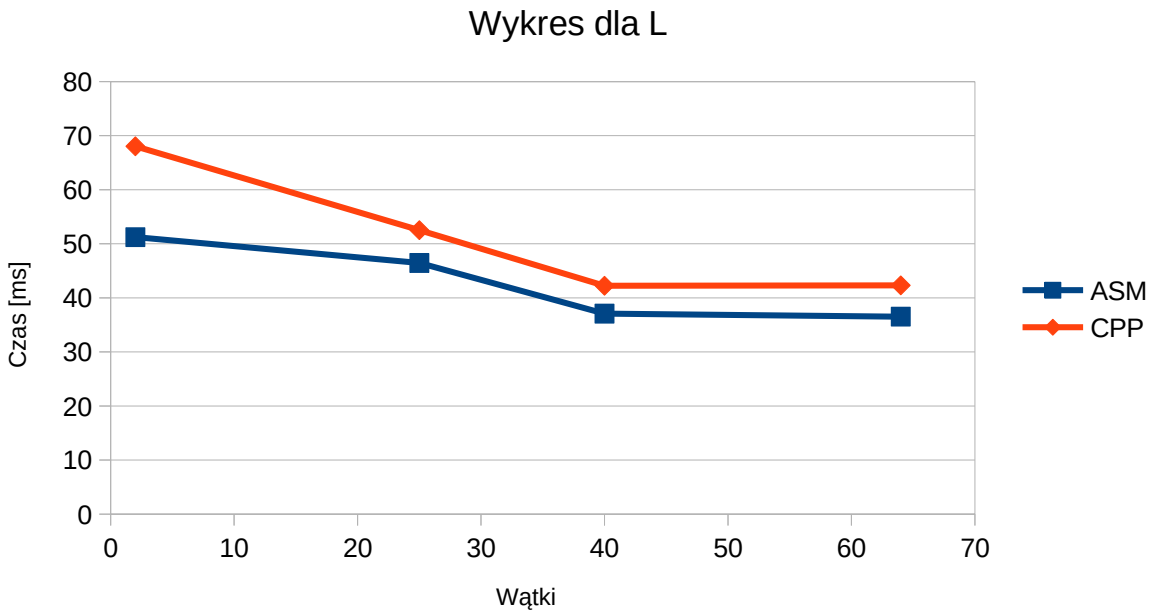
Operacje wykonane na trzech zestawach danych:

	S		M		L	
wątki	ASM	CPP	ASM	CPP	ASM	CPP
2	6,7793	6,3943	28,1213	32,1141	51,2215	68,0395
25	6,4541	6,2773	23,2276	17,0354	46,4412	52,5172
40	6,4143	5,8389	21,6544	10,7581	37,0827	42,2369
64	5,8941	5,9842	22,9456	14,5972	36,5249	42,3133

Powyższe dane przedstawione na wykresach:







Na podstawie tych testów, ciężko wysnuć jedną konkretną zależność. Dla zestawów S i M zazwyczaj lepszy czas dawała biblitioteka CPP. Dla obrazu L nieznacznie lepszy czas był dla biblioteki ASM. Lepszy czas wykonania CPP może wynikać z implementacji rozkazu __mm_set1__epi8. Jest to jedyna różnica między procedurami, prawdopodobnie w CPP jest ona zaimplementowana w lepszy sposób niż za pomocą przesunięcia bitowego i dodawania. Można również zauważyć, że czas wykonania przy użyciu 2 wątków – czyli ilości wątków wykrytych przez program nie jest optymalny, i przy zwiększeniu tej ilości, czas maleje zarówno dla ASM jak i CPP.

4. Analiza i porównanie czasowe

Wykonanie tego projektu było ciekawym i pouczającym doświadczeniem. Był to tak naprawdę mój pierwszy w stuprocentach wykonany samodzielnie program w assemblerze. Również było to moje pierwsze zetknięcie z operacjami wektorowymi, zarówno w c++, jak i assemblerze. W moim odczuciu mają one dość wysoki próg wejścia, a w internecie nie ma za dużo informacji na ich temat. Podczas realizacji tego projektu dowiedziałem się wiele nowych rzeczy, np. jak dokładnie przekazuje się argumenty do procedury zaimplementowanej w asm. Co ciekawe, trudniejsze było dla mnie napisanie procedury w c++. Instrukcje wektorowe w tym języku są dość specyficzne, a zmienne typu _128i są tak naprawdę rejestrami. Również pierwszy raz pisałem GUI w bibliotece Windows Forms, jednak uważam tą część za najprostszą, biliotaeka ta jest bardzo intuicyjna i ten etap wykonałem najszybciej. Kolejną trudnością tego projektu był brak możliwości debugowania bibliotek w formacie .dll. Ten fakt znacząco przedłużył czas napisania działającego programu.