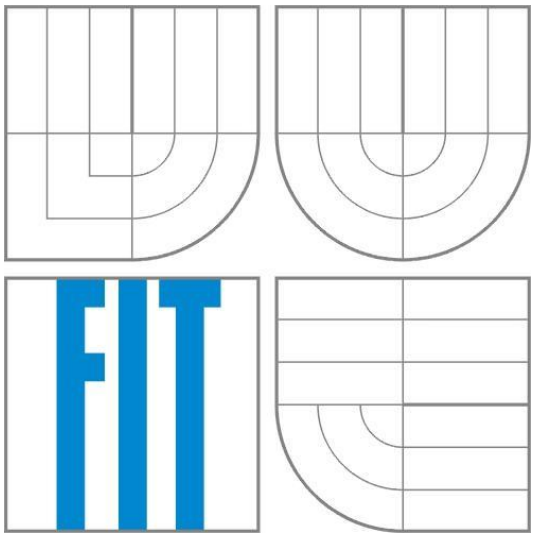
Fakulta Informačních Technologií

Vysoké Učení Technické v Brně



Dokumentácia k semestrálnemu projektu pre predmet

Pokročilé Asemblery

Varianta zadania: Projekt typu Benchmark - Rozostření

Autor: Martin Fajčík

Dátum: 22.12.2013

Obsah

1 Úvod

1.1 O dokumente

Tento dokument vznikol ako technická dokumentácia k semestrálnemu projektu pre voliteľný predmet IPA (Pokročilé Asemblery), ktorého úlohou v mojom prípade bola implementácia algoritmu pre rozostrenie obrazu (*Gaussovské rozmazanie, tzv. Gaussian Blur*) v jazyku asembler (pomocou prekladaču MASM vstavaného v odporučanom vývojovom prostredí Visual Studio), alebo pomocou intrinsic funkcií, jeho zrýchlenie za pomoci SSE(Streaming SIMD Extensions) a MMX (Multimedia Extensions) inštrukcií oproti referenčnej implementácii a taktiež dokumentácia.

Dokumentácia si kladie za ciel oboznámiť čitateľa so zvoleným spôsobom implementácie, so štruktúrou programu, s optimalizáciami a taktiež s implementačnými problémami, s ktorými som sa v priebehu riešenia tohto projektu stretol.

1.2 Štruktúra dokumentu

Dokument je rozdelený na kapitoly a podkapitoly, ktoré odpovedajú riešeniu jednotlivých častí projektu, ktorými je inicializácia potrebných konštánt a premenných, algoritmus pre filter ,ktorý rozmazáva vstupný obrázok v bitmapovom formáte *(.bmp*) vertikálne a algoritmus pre filter, ktorý rozmazáva vstupný obrázok horizontálne. Okrem samotnej implementácie je priestor vyhradený aj pre jednotlivé problémy, s ktorými som sa pri implementácii stretol. Na záver je v samostatnej kapitole zhodnotený výsledok riešenia projektu a jeho prínos.

2 Proces implementácie

2.1 Predpísaný prospekt

Súčasťou zadania projektu typu Benchmark bol súbor *Benchmark2011 v2.zip* , ktorý obsahoval projekt Benchmark2011.*sln* určený pre program *Microsoft Visual Studio* a ktorého súčasťou boli už predpísané referenčné riešenia zadaní typu Benchmark napísaných a optimalizovaných v jazyku C++. Okrem toho tento projekt obsahoval taktiež periférie určené k testovaniu projektu, určeniu efektívnosti a chybovosti nášho riešenia oproti referenčnému riešeniu ako aj samotné určenie toho či projekt pracuje správne alebo nesprávne. Mojou úlohou bolo doplniť správne riešenie do súboru *gaussian_blur.cpp*.

2.2 Inicializácia programu

K správe programu sme mali k dispozícii v súbore *gaussian_blur.cpp* predpísané funkcie, ktoré boli v jazyku C++ zastrešené pod triedou:

```
class CGaussianBlur : public CAlgorithm
```

Pričom dve z týchto funkcií slúžili priamo pre inicializáciu, v respektíve de-inicializáciu zdrojov, konštánt a premenných.

```
void CGaussianBlur::CustomInit()
```

```
void CGaussianBlur::CustomDeinit()
```

Použitie týchto funkcií bolo nepovinné a práve preto som sa rozhodol v rámci či už efektivity (zbytočné prenášanie konštánt z jednej funkcie do druhej), ako aj zjednodušenia, keďže jazyk C++ zatiaľ dostatočne nepoznám inicializovať konštanty a premenné priamo vo funkcii určenej pre algoritmus.

```
void CGaussianBlur::Blur(const unsigned char *image, int image_w, int image_h, unsigned char *new_image, float user_radius)
```

```
void CGaussianBlur::Blur(const unsigned char *image, int image_w, int image_h, unsigned char *new_image, float user_radius)
```

(funkcia blur)

2.21 Parametre funkcie Blur

Táto podkapitola slúži pre stručný prehľad významu parametrov, ktoré boli predefinované pre funkciu Blur, ktorá sa nachádza v triede CGaussianBlur.

- `const unsigned char *image`
  - Je ukazateľ do poľa bezznamienkových dátových typov char (1 byte)
  - Reprezentuje sekvenciu pixelov zostavených z obrázku, ktorý je vstupom programu
- `int image_w`
  - Reprezentuje šírku (width) obrázku pomocou dátového typu integer (4 byty)
- `int image_h`
  - Reprezentuje výšku(height) obrázku pomocou dátového typu integer (4 byty)
- `float user_radius`
  - Zastupuje veľkosť s akou je rozmazávanie vykonávané, dátový typ float (4 byty)
- `unsigned char *new_image`
  - Výstupné pole pixelov spracovaných pomocou Gaussovského filtru, bezznamienkový dátový typ char (1 byte)

2.22 Konštanty a masky

Konštanty v mojom programe sú väčšinou použité vo formách masiek pre robenie či už logických, tak aj matematických operácií. Najmä spočiatku, keď som ešte nepoznal množstvo trikov pre nahratie určitých špecifických konštánt som používal masky, ktoré pre nedostatok času pre refaktorizáciu a úpravu v programe ostali. Bolo tiež dôležité správne zarovnať veľkosť ukazateľov, čo som dosiahol rozšírením atribútu popisujúceho konštanty a premenné o kľúčové slovo *__declspec* s modifikátorom align(#). Výsledná deklarácia jednotlivých masiek teda vyzerala približne takto:

```
__declspec(align(16)) const static long _maskallnegfloat[4] = {0x80000000,0x80000000,0x80000000,0x80000000};
```

(maska pri logickej operácii AND otáča znamienko u desatinných čísel s jednoduchou presnosťou v XMM registri)

Neskôr som začal používať operácie, ktoré som si niekoľkokrát všimol na cvičeniach a pomocou ktorých vytvorím masky oveľa rýchlejšie bez zbytočného prístupu do pamäte.

```
pcmpeqw xmm1,xmm1;//loads 0.5
pslld xmm1,26
psrld xmm1,2
```

(operácia nahrá 4x číslo 0.5 do XMM registru)

```
pcmpeqw xmm1,xmm1;//loads 1
pslld xmm1,25
psrld xmm1,2
```

(operácia nahrá 4x číslo 1.0 do XMM registru)

2.23 Premenné, zdroje a úvodné výpočty

Vo veľkej časti projektu sa premenné nachádzajú len minimálne, keďže ich existencia je simulovaná pomocou XMM a MMX registrov, s ktorými je akákoľvek manipulácia oveľa rýchlejšia. Výnimku tvoria úvodné výpočty, ktoré z hľadiska väčšej prehľadnosti a minimálnom rozdiele z hľadiska efektivity výhodnejšie. V ich prípade som si zvolil pomocnú premennú dátového typu `__m128` c, ktorá by v prípade použitia inline asembleru bola nahradená registrom. Použitie intrinsic inštrukcií bolo taktiež vhodné vďaka tomu, že obsahujú funkciu(makro) `_mm_floor_ss` ,ktorej inline asembler varianta neexistuje.

Pre výpočet a úschovu začiatočnej a koncovej pozície, podľa ktorej sa neskôr kontroloval cyklus filtrujúci obrázok a rádius, ktorý určoval silu rozmazania som použil MMX inštrukcie a premenné typu `__m64 radius, startpos, endpos` .

Podobne ako v prípade demonštrovanom referenčnou implementáciou, aj vo svojej implementácii som potreboval priestor pre úschovu hodnôt jednotlivých pixelov. Tento priestor som vytvoril pomocou C++ operácie `new`, pomocou ktorej som za ukazateľom `__m128 *buffer` alokoval image_h alebo image_w (podľa toho, ktorý rozmer bol väčší) položiek typu `__m128`.

```
__declspec(align(16)) static __m128 *buffer = new __m128[(image_h > image_w)? image_h : image_w];
```

2.3 Vertikálne rozostrenie

Podobne ako horizontálne rozostrenie, aj vertikálne rozostrenie je implementované ,vďaka lepšej schopnosti optimalizácie, v inline asembleri. Optimalizácia tejto časti programu tvorila najťažší oriešok, pretože bola zároveň aj veľkým chytákom. Spočiatku som sa snažil optimalizovať vnútorný cyklus, no po pár dňoch práce som si uvedomil, že hoci sa cez podmienky dokážem dostať pomocou masiek, vďaka premennej dif, ktorá menila hodnotu prvku ďalšieho cyklu bola optimalizácia pomocou paralelného spracovania štyroch prvkov vnútorného cyklu zaraz nemožná. Až po vyriešení tohto problému som sa dokázal pustiť do zdanlivo nie až tak efektívnej a možnej optimalizácie vonkajšieho cyklu.

2.3.1 Optimalizácie

Jednou z vecí, ktorú som odstránil, pretože bola v referenčnom riešení zbytočná bola časť inicializujúca sumu vo vonkajšom cykle. Všimol som si, že časť `(y_start - radius - 1)*image_w` sa vždy vyhodnotí ako 0, keďže y_start sa v priebehu programu nemení a platí:
`int y_start = radius + 1.`

Ďalšou optimalizáciou bola simulácia niektorých premenných pomocou voľných registrov.

Premenná v referenčnom riešení	Premenná simulovaná registrom
sum	XMM5
dif	XMM6
p	XMM7
x	EAX
y	ECX

(pozn. ostatné premenné použité v cykle sú pomenované rovnako ako v referenčnej kópii)

Ďalšou optimalizáciou je v tejto časti bolo už spomínané použitie SSE registrov a príslušných inštrukcií pre výpočet štyroch x-súradníc naraz. Jedným s problémov, na ktorý som narazil bolo však práve prekračovanie medzí obrázku práve kvôli paralelnému spracovaniu pri nevhodnej šírke. To bolo korektne ošetrené podmienkou, ktorá zaručuje že takéto prípade sa jednoducho preskočia.

Poslednou z rád optimalizácií, ktoré by som rád spomenul je využitie komplexnejších inštrukcií SSE4.x, ako napríklad použitie inštrukcie *pmovzxbd* pre nahratie štyroch prvkov s poľa unsigned charov do XMM registru.

2.4 Horizontálne rozostrenie

Podobne ako vertikálne rozostrenie, aj horizontálne rozostrenie je implementované ,vďaka lepšej schopnosti optimalizácie, v inline asembleri. Pri riešení tejto časti som sa stretol hlavne s problémami, ktoré sa nachádzali aj v predchádzajúcej časti 2.3. avšak teraz už som vedel ako ich treba vyriešiť. Táto časť riešenia je, podobne ako aj časť 2.3.,riešenia optimalizáciou vonkajšieho cyklu.

2.4.1 Optimalizácie a problémy

Narazil som taktiež na podobný prípad zbytočnosti kódu, ktorý som vo svojom riešení vynechal.

```
float sum = (float)new_image[x_start - radius - 1]...
        (červeno zvýraznená časť je zbytočná, keďže platí int x_start = radius + 1)
```

Pri získavaní ďalšieho pixelu p som oproti vertikálnemu rozostreniu (2.3) narazil na problém, ktorý zapríčiňoval to, že štyri pixely, ktoré som chcel načítať neležali za sebou v pamäti a tak bolo potrebné tieto pixely načítať zvlášť po jednom a „umelo“ načítať do XMM registru. Podobný problém sa taktiež vyskytol u zápisu do obrázku, kedy program musel zapisovať štyri prefiltrované pixely do pamäti na rôzne miesta.

```
float p = (float)new_image[x + radius + y0,1,2,3*image_w];
new_image[x + y0,1,2,3*image_w] = ...
```

Premenná v referenčnom riešení	Premenná simulovaná registrom
sum	XMM5
dif	XMM6
p	XMM7
y	EAX
x	ECX

(pozn. ostatné premenné použité v cykle sú pomenované rovnako ako v referenčnej kópii)

V neposlednom rade, aj v tejto časti boli použité inštrukcie pre rýchle nahratie konštánt do potrebných registrov, alebo komplexné SSE4.x inštrukcie spomenuté už v kapitolách 2.2, 2.3.

3 Záver

Na projekte som pracoval, keďže to bol môj prvý projekt v asembleri vôbec, 8 dní , približne 60-70 hodín čistého času. Našťastie to prinieslo svoje ovocie a rýchlosť programu sa oproti referenčnej implementácii zrýchlila (priemerne) 4.5x, pričom z testovania vyplýva, že nedochádza k žiadnemu rozdielu v pixeli medzi referenčnou a mojou implementáciou.

```
Total difference from the reference image = 0.000000
Total error per pixel = 0.000000
Success!!! Algorithm [Gaussian blur] succeeded in the conformance testing.
Average performance of the reference algorithm: 14.724759 ms, 32998 cycles.
Average performance of your algorithm: 3.088959 ms, 6922 cycles.
You have managed to increase performance by 4.766900
Press any key to close the application.
```

Napriek tomu čas investovaný do tohto projektu by som zhodnotil ako z mojej strany pozitívnu, i keď náročnú skúsenosť, pretože práve skúsenosť z tohto projektu mi pomohla dostatočne pochopiť nie len funkčnosť ale aj využitie technológií SSE a MMX . Pevne verím, že v budúcnosti bude formu tohto projektu možné riešiť aj pomocou ešte rýchlejšej technológie AVX

4 Prílohy

Ukážky použitia Gausského rozostrenia