Využitie špeciálnych inštrukcií procesora x86 pri vykresľovaní Mandelbrotovej množiny

Bc. Dávid Beňo

***Abstract*—V tomto článku sme vysvetlili čo je to paralelizmus na úrovni inštrukcií, možnosti ako paralelizovať vykonávanie jednej inštrukcie na viacerých tokoch dát a tým zrýchliť vykonávanie časovo náročných programov. Rozobrali sme hlavné rozšírenia procesorov firmy Intel x86 a možnosti využitia špeciálnych inštrukcií pre prácu s týmito rozšíreniami. Ďalej sme vysvetlili čo je to fraktál a aká je to Mandelbrotova množina. Na zvolenej architektúre x86 sme vytvorili program pre generovanie obrázkov Mandelbrotovej množiny pomocou špeciálnych inštrukcií SSE a AVX. Meraním rýchlosti vykreslenia rovnakého obrázku rôznymi spôsobmi (bez použitia špeciálnych inštrukcii aj s ich použitím) sme zistili zrýchlenie a tým pádom aj užitočnosť daných rozšírení.**

Kľúčové slová—špeciálne inštrukcie, rozšírenia procesora, sse, avx, mandelbrot

I. Úvod

Neustále zvyšovanie výkonu procesorov narazilo na isté limity. Nie sme schopní donekonečna vyvíjať procesory, ktoré pracujú na stále vyššej frekvencii. Preto sa pri moderných procesoroch hľadá ďalší výkon v paralelizme výpočtových úloh. Známejšou formou paralelizmu je paralelizmus úloh, ktorý je na hardvérovej úrovni podporovaný viacerými jadrami, hyperthreadingom alebo špeciálnymi inštrukciami podporujúcimi multitasking v operačnom systéme. Všetky tieto formy paralelizmu ale fungujú v základnom nastavení takým spôsobom, že pri výpočtoch v daných úlohách každá inštrukcia sa vykonáva iba nad jedným tokom dát. Tento prístup sa volá SISD - "single instruction, single data" a zodpovedá ešte von Neumanovej architektúre zo štyridsiatych rokov minulého storočia. Inštrukcia z pamäte číta alebo do pamäte zapisuje vždy len jednu hodnotu. Práve prístup do pamäte je časovo veľmi drahý. Tento problém rieši iný prístup paralelizmu a tým je SIMD - "single instruction, multiple data" a teda, že nad viacerými tokmi dát sa naraz vykoná rovnaká inštrukcia. Je to síce menej známa forma paralelizmu ale napriek tomu je veľmi účinná. Procesor dokáže naraz z pamäte prečítať vektor dát, vykonať nad všetkými položkami vektora rovnakú inštrukciu a výstupom je tiež vektor dát. Tento spôsob paralelizmu ale potrebuje explicitnú podporu zo strany programátora, ktorý musí program pre takéto výpočty vytvoriť a nemôže sa len spoľahnúť na kompilátor.[1, 2, 3]

II. SIMD – Single Instruction, Multiple Data

Flynnová klasifikácia je zrejme najznámejšia klasifikácia paralelných systémov, vznikla v roku 1966. Systémy sú delené z dvoch hľadísk – toku inštrukcií a toku dát. Táto klasifikácia obsahuje štyri hlavné typy paralelných systémov:

	Single Instruction	**Multiple Instruction**
Single Data	*SISD*	*MISD*
Multiple Data	*SIMD*	*MIMD*

Fig. 1. Tabuľka Flynovej klasifikácie

Práve myšlienka SIMD a teda vykonávanie jednej inštrukcie nad viacerými tokmi dát bola námetom pre vznik viacero procesorových architektúr alebo rozšírení, ktoré by toto zabezpečili.[6]

Motivácia, prečo sa vývoj uberá týmto smerom je ten, že pomocou SIMD prístupu vieme značne urýchliť výpočty algoritmov, ktoré sú časovo náročné. Existujú výpočtove úlohy, ktorých skalárne spracovanie údajov vieme prerobiť do vektorového spracovania. Najčastejšie sa môžeme s úlohami, ktoré sa dajú pralelizovať spôsobom SIMD v týchto oblastiach:

Grafika – aplikácie filtorv, pixel shaders, ray tracing.

Fyzika – Výpočty medzi množstvom častíc (gravitácia).

Matematika – Operácie s maticami (Neurónové siete), analytická geomeria

Vo všeobecnosti ale platí, že SIMD prístup sa dá aplikovať na akúkoľvek úlohu, pri ktorej sa nad väčším množstvom dát vykonáva rovnaká inštrukcia a tým sa dá značne urýchliť celý výpočet.

A. *VLIW – Very Long Instruction Word*

Prvou takouto architektúrou, ktorá sa pokúša spracovať údaje metódou SIMD, je VLIW - procesorová architektúra, ktorá umožňuje inštrukčný paralelizmus. Toto je dosiahnuté tým, že VLIW používa úplne iný formát inštrukčnej sady napríklad na rozdiel od procesorov CISC. Tie majú premennú dĺžku inštrukčných slov ale VLIW používa podobne ako procesory RISC pevnú dĺžku inštrukcie. Avšak dĺžka inštrukcií je v tomto prípade omnoho dlhšia rádovo až niekoľko stoviek bitov. Je tomu tak preto, lebo každé inštrukčné slovo je rozdelené na viacero polí, pričom každé obsahuje kód jednej operácie. Dĺžka týchto polí môže byť pevná ale aj premenná. Procesor potom môže paralelne spustiť všetky úlohy, ktoré sú zakódované v poliach inštrukčného slova. Keďže je zlučovanie operácií do inštrukčného slova záležitosťou kompilátoru,

nedeje sa žiadna kontrola či dané operácie skutočne môžu byť spustené paralelne. Procesor sa tiež nemusí snažiť rozdeliť operácie do pipeline, pretože aj to je určené vopred kompilátorom – konkrétne pozíciou operácie v inštrukčnom slove. Vďaka zložitej predpríprave na strane kompilátoru je procesor VLIW na hardvérovej úrovni jednoduchý a teda aj lacný. Na druhú stranu ak sa kompilátoru nebude dariť z rôznych dôvodov ukladať do inštrukčného slova operácie, ktoré môžu byť vykonané paralelne (polia budú väčšinou obsahovať len inštrukciu NOP), drasticky sa zníži výkon procesoru. V praxi to môže znamenať komerčný neúspech takéhoto procesoru. [11]

B. EPIC – Explicitly Parallel Instruction Computing

Táto architektúra procesora vychádza z architektúry VLIW a vylepšuje ho ďalšími softvérovými postupmi spracovania inštrukcií ako je napríklad špekulatívne načítanie. Na základe EPIC bola firmami Intel a HP vytvorená architektúra 64 bitového procesoru IA-64 a bola nasadená v procesoroch Itanium a Itanium2. Avšak tie sa nestretli s komerčným úspechom.

C. X86 – Špeciálne registre

Ďalším možným spôsobom ako dosiahnuť SIMD funkcionalitu procesoru aj na už existujúcej architektúre je použitie špeciálnych veľkých registrov. Tieto registre môžu v sebe držať vektor dát, ktorého veľkosť závisí od veľkosti registra a môže sa vykonávať výpočet medzi viacerými takýmito registrami.

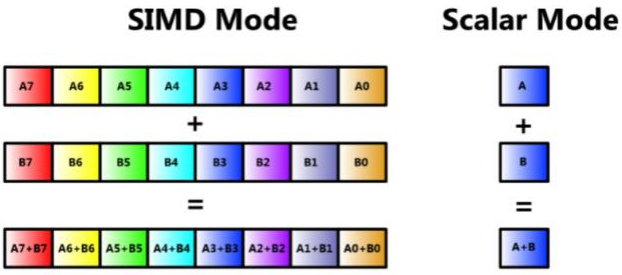

Fig. 2. Obrázok vektorového a skalárneho spracovania dát[3]

Vďaka tomu znížime počet prístupov do pamäte a vieme naraz pracovať s väčším množstvom dát na rozdiel od skalárneho prístupu, ktorý musí iterovať po jednotlivých položkách.[3]

III. Rozšírenia Procesorov Intel x86

Firma Intel, ktorá vyrába procesory hlavne architektúry x86 sa tiež pokúsila o ich rozšírenie o prvky, ktoré by zabezpečovali SIMD spracovanie dát. Svoje procesory vybavuje rôznymi špeciálnymi registrami, ktoré sú väčšie ako klasické registre a môžu sa použiť na paralelné spracovanie dát. Použitie týchto registrov v rámci programu nezabezpečuje kompilátor ale samotný programátor.

A. MMX – Multi Media Extension

Toto riešenie bolo uvedené na trh v roku 1997 pre vtedajšie procesory Pentium. Spočívalo v tom, že procesor okrem klasických 32 bitových registrov obsahoval aj ďalšie väčšie registre. Jedná sa konkrétne o osem 64 bitových registrov, ktoré sa označovali MM0 – MM7. Každý z týchto registrov mohol v sebe držať jeden integer o veľkosti 64 bitov alebo viacero menších integerov (dva 32 bitové, štyri 16 bitove alebo osem 8 bitových). Toto riešenie však podliehalo aj obmedzeniam. Keďže MMX registre boli mapované do registrov FPU, ktoré sa využívali na operácie s číslami s pohyblivou desatinnou čiarkou, nebolo možné naraz vykonávať operácie aj s celými číslami. Po použití MMX registrov bolo nutné výsledok uložiť do pamäte ešte pred vykonaním operácie s číslami s pohyblivou čiarkou. Na toto nebezpečenstvo neupozorňoval ani kompilátor, takže za správny chod programu zodpovedal programátor. Pre vytvorenie programového kódu bola vytvorená knižnica, ktorá obsahovala 57 inštrukcií, pre prácu s MMX registrami.[9]

B. SSE – Streaming SIMD Extensions

1) SSE

V roku 1999 Intel navrhol pre svoj procesor Pentium III novú inštrukčnú sadu SSE ako odpoveď na podobnú sadu inštrukcii od konkurenčného AMD 3DNow! a zároveň ako vylepšenie MMX. Rozšírenie spočívalo hlavne v nových ôsmich 128 bitových registroch, ktoré mohli byť adresované priamo s označením XMM0 – XMM7. Navyše toto rozšírenie tiež vyriešilo problém MMX, kde sa dalo pracovať naraz len s jedným typom dát, a teda vďaka SSE sa mohli vyvíjať programy, ktoré kombinovali celé čísla aj čísla s pohyblivou desatinnou čiarkou a nepotrebovali žiadne špeciálne zaobchádzanie. Avšak je potrebná istá podpora od operačného systému, pretože pri preplánovaní úloh musia byť zálohované. V rámci rozšírenia SSE pribudol aj nový kontrolný register MXCSR o veľkosti 32 bitov. Táto nová sada obsahovala 70 inštrukcii, ktoré boli dostupné programátorovi na prácu so špeciálnymi registrami. Z toho 50 nových inštrukcií pre prácu s vektorovými aj skalárnymi operáciami na číslach s pohyblivou desatinnou čiarkou. 8 nových inštrukcií pre ovládanie "cache-ovania" všetkých MMX vrátane streamovania dát do pamäte. A 12 nových inštrukcií, ktoré rozširovali MMX inštrukčnú sadu.[9]

2) SSE2

Ďalšie rozšírenie prišlo na trh v roku 2001 vo podobne SSE2, ktoré rozširovalo prechádzajúce SSE a pridalo nových 144 inštrukcií. Toto rozšírenie bolo spočiatku použité v procesoroch Intel Pentium IV a Xeon, neskôr sa prenieslo aj na procesory AMD. Počet registrov ani ich bitová dĺžka sa nezmenila ale hlavnou zmenou bola interná štruktúra vektoru. Každý z týchto registrov XMM0 – XMM7 mohol reprezentovať:[3]

- Štyri 32 bitové čísla s pohyblivou rádovou čiarkou
- Dve 64 bitové čísla s pohyblivou rádovou čiarkou
- Dve 64 bitové celé čísla
- Štyri 32 bitové celé čísla
- Osem 16 bitových pre krátke celé čísla
- Šestnásť 8 bitových registrov pre bajty alebo znaky

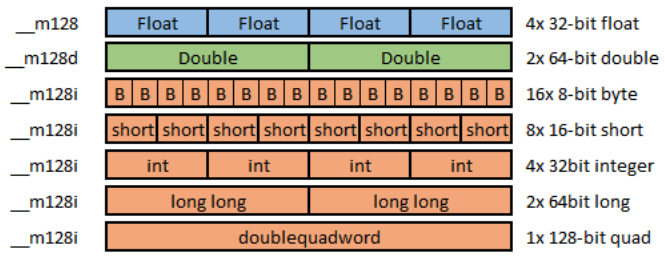

Fig. 3. Možnosti prístupu k SSE2 registru

3) SSE3

Intel v roku 2004 vydal ďalšie rozšírenie sady inštrukcií SSE. Toto rozšírenie pridalo len 13 nových inštrukcií ale dovolilo nové možnosti spracovania vektorov. Vďaka tomuto rozšíreniu sa dali spracovávať čísla horizontálne v rámci jedného vektora a nie len vertikálne medzi viacerými vektormi. Tiež boli pridané kontrolné inštrukcie pre lepšie fungovanie s Intel HyperThreading-om.

4) SSE4

Posledným zo série SSE rozšírení je SSE4, s ktorým prišla firma Intel v roku 2006. Toto rozšírenie sa ale skladá z dvoch podmnožín a síce SSE4.1 a SSE4.2. Tieto dokopy priniesli 54 nových inštrukcií (SSE4.1 – 47 a SSE4.2 - 7).

C. AVX – Advanced Vector Extension

1) AVX

Toto rozšírenie prišlo na procesoroch Intel radu Sandy Bridge v roku 2011 a bolo to prvé rozšírenie inštrukčnej sady po niekoľkých rokoch. Prinieslo 16 nových registrov o veľkosti 256 bitov, ktoré sa označovajú YMM0 – YMM15. Okrem týchto tiež obsahuje 32 bitový kontrolný register MXCSR. YMM registre sú mapované cez staršie 128 bitové XMM používané pri SSE inštrukciách a berú ich ako svoju polovičnú časť. Graficky to je znázornené na obrázku.[3]

Ku každému z týchto YMM registrov môžeme pristupovať buď ako ku štyrom 64 bitovým double číslam alebo ako ku ôsmim 32 bitovým číslam s pohyblivou desatinnou čiarkou.

AVX uviedlo novú kódovaciu schému VEX, ktorá rozširovala opcode pre budúce inštrukcie a zabezpečovala možnosť práce s 256 bitovými registrami. Vďaka VEX vznikol aj troj-operandový formát inštrukcie, kde cieľová adresa je odlišná ako dva operandy. Pri staršom SSE sa používal dvoj-operandový režim kde a = a + b. Vďaka AVX sa oba zdrojové operandy uchovajú, pretože dokážeme použiť nový režim v tvare c = a + b. Tento režim je ale limitovaný pre registre YMM a nie je použiteľný pre všeobecné registre (napr. EAX).[3]

2) AVX2

Toto rozšírenie predstavené pri procesoroch Intel Haswell, dopĺňa staršie AVX o niekoľko nových funkcionalít. Napríklad rozširuje vektorové inštrukcie pre prácu s celým číslom na 256 bitov alebo posun vektorov.[10]

3) AVX-512

Je to rozšírenie z roku 2015 YMM registrov na dĺžku 512 bitov a pridáva ďalších 16 registrov a teda ich celkový počet sa zvýšil na 32, označujú sa ZMM0 – ZMM31. AVX-512 inštrukcie sú kódované novým EVEX prefixom. Ten napríklad umožňuje používať až 4 operandy alebo tiež poskytuje skalárny pamäťový mód s automatickým broadcastom.

D. Príklad použitia špeciálnych inštrukcií

Všetky inštrukcie typu SSE alebo AVX sú v súčasnosti pre jazyk C/C++ podporované natívnymi knižnicami. Pre SSE to je knižnica <xmmintrin.h>. Pre AVX sa jedná o knižnicu <immintrin.h> a kompilátor GCC ju podporuje od verzie 4.4.

1) Numerické operácie

Jednou zo základných operácii je sčítanie (násobenie, ...). Rovnaká inštrukcia sa vykoná paralelne nad každou dvojicou údajov vektorov a výsledkom je vektor so sčítanými hodnotami.

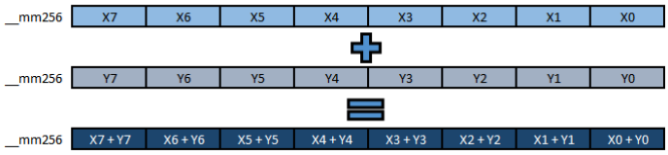

Fig. 4. Súčet vektorov

Vy koná sa operácia:

FOR j := 0 to 7

*i := j*32*

dst[i+31:i] := a[i+31:i] + b[i+31:i]

ENDFOR

dst[MAX:256] := 0

[5]

2) Broadcast

Ďalší prípad práce s registrami nastane, keď chceme nastaviť pre každý údaj vo vektore rovnakú hodnotu. Inštrukčná sada pre tento prípad poskytuje riešenie. Napríklad pri nastavení čísla s pohyblivou desatinnou čiarkou s hodnotou *a* pre všetky elementy vektoru, použijeme inštrukciu *__m256 _mm256_set1_ps (float a)*. Pseudokód vykonanej operácie:

FOR j := 0 to 7

*i := j*32*

dst[i+31:i] := a[31:0]

ENDFOR

dst[MAX:256] := 0

[5]

3) Maska

Často sa môže stať, že je potrebné vykonať výpočet len s jednou časťou vektoru. Pre tento prípad sa používa maska. Maska je výsledkom logickej operácie medzi vektormi. Má mnoho podobností s dátovým typom boolean (sú výsledkom logických operácií na jednotlivých číslach alebo iných booleanoch), ale vnútorne každá časť masky musí mať všetky

bity 0 alebo 1. Môžu byť využité napríklad pri načítaní hodnôt do vektoru.

Napríklad ak chceme mať v nejakom vektore len hodnoty vačšie ako 3. Porovnáme dva vektory a vytvoríme masku:

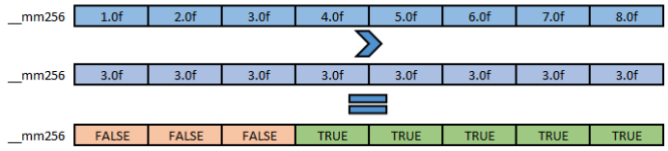

Fig. 5. Porovnanie vektorov

Následne vykonáme podmienený výber z oboch vektorov a vznikne vektor, v ktorom sú len čísla väčšie ako 3:

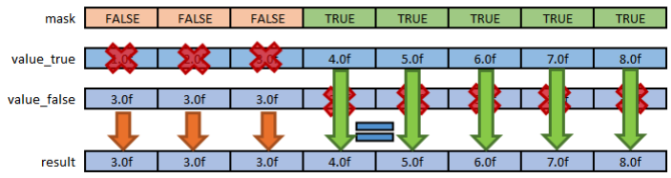

Fig. 6. Podmienený výber údajov z vektorov

Maska môže byť tiež využitá pri čítaní alebo zapisovaní údajov do vektora z pamäte alebo opačne. Napríklad pri zapisovaní do pamäte, môžeme použiť procedúru void _mm256_maskstore_ps (float * mem_addr, __m256i mask, __m256 a). Vykoná sa operácia:

FOR j := 0 to 7

*i := j*32*

IF mask[i+31]

MEM[mem_addr+i+31:mem_addr+i]:=a[i+31:i]

FI

ENDFOR

[5]

IV. Výpočtová Úloha

Pre demonštráciu fungovania špeciálnych inštrukcií a registrov sme si zvolili výpočet Mandelbrotovho fraktálu. Program je písaný v jazyku C a má tri verzie. Prvá nepoužíva špeciálne inštrukcie, druhá používa SSE registre s dĺžkou 128 bitov a tretia verzia používa inštrukcie AVX s dĺžkou registrov 256 bitov. Porovnaním časov výpočtu rovnakej množiny pixelov dokážeme pridanú hodnotu procesorových rozšírení.

A. *Fraktály*

Je to geometrický objekt, ktorý je nepravidelný, fragmentovaný, fragmentovaný geometrický tvar, ktorý môže byť rozdelený na časti, z ktorých je každá aspoň približne podobná, zmenšená kópia celého geometrického tvaru. Táto vlastnosť tiež býva nazývaná sebepodobnosť. Fraktály sú definované pomerne krátkou rekurzívnou definíciou, ako zakresliť ich body nad množinou komplexných čísel. Ide o objekt, ktorého Hausdorffova miera je väčšia než topologická dimenzia. To znamená, že fraktál nemá ako kocka 3 dimenzie (rozmery), ale jeho dimenzia je zväčša neceločíselná. Obsah fraktálov (resp. objem) je konečný, no ich obvod (resp. povrch) je nekonečný. Ide o jedny z najzložitejších geometrických objektov, ktoré súčasná matematika skúma a majú často prekvapivo jednoduchú matematickú štruktúru. Medzi najznámejšie fraktály patria Mandelbrotova alebo Juliova množina. [12]

B. *Mandelbrotov fraktál*

Mandelbrotova množina (pomenovaná po matematikovi Benoitovi Mandelbrotovi) je jedným z najznámejších fraktálov. Je definovaná ako množina komplexných čísel c pre ktoré platí:

$$\lim_{n\to\infty} |z_n| \neq \infty$$

kde pre postupnosť Z_0, Z_1, Z_2, ... je definovaná rekurzívnym predpisom:

$$z_0 = 0; \qquad z_{n+1} = z_n^2 + c$$

Bod *c* teda patrí do Mandelbrotovej množiny práve vtedy, ak uvedená limita neexistuje, alebo je konečná (napr. c = 0). Je možné jednoducho dokázať, že postupnosť ide do komplexného nekonečna pre všetky |c|>2, takže ak ktorýkoľvek člen postupnosti prekročí túto absolútnu hodnotu, potom *c* nie je prvkom Mandelbrotovej množiny. Celá množina leží vo vnútri kruhu so stredom v počiatku súradnicovej sústavy a polomerom 2.

Pri praktickej implementácii sa pre každý bod rovnica opakovane vyčísľuje a vo chvíli, keď |zn| > 2, je zrejmé, že pre daný bod bude rovnica divergovať (a pri grafickom zobrazovaní sa táto hodnota n, pre ktorú bod túto hranicu prekročil, spravidla prevádza na farbu). Ak ani po dopredu zvolenom počte iterácii k prekročeniu tejto hranice nedôjde, je bod považovaný za súčasť Mandelbrotovej množiny. Nastavenie tejto hranice ovplyvňuje výsledný obrázok: pre príliš malú hodnotu budú niektoré body nesprávne označené ako patriace do množiny, ale veľký počet iterácii vyžaduje dlhší čas výpočtu.[12]

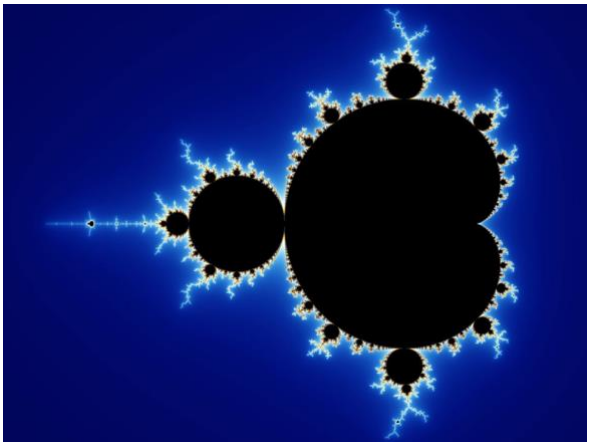

Fig. 7. Obrázok vykreslenej Mandelbrotovej množiny

C. *Algoritmus výpočtu*

Vzorec $Z = Z^2 + C$ musíme aplikovať na všetky body v množine. Tá je v rozsahu X od -2 do 2 a Y tiež od -2 do 2. Budeme teda potrebovať 2 do seba vnorené cykly a 2 celočíselné premenné X a Y, ktoré nám tieto body vypočítajú.

Určíme si nejaký počet pokusov (iterácií) na vyskúšanie, či práve testovaný bod do množiny patrí alebo nie. Vytvoríme teda premennú alebo konštantu *max_iter*, ktorej priradíme napríklad hodnotu 20. Ďalej nejaké počítadlo pokusov, ktoré si nazveme iter. K samotnému testu bodu budeme potrebovať komplexné číslo Z, ktoré pred každým pokusom vynulujeme, komplex. číslo C (reálna a imaginárna časť budú premenné X a Y, pretože C je pozícia bodu v množine) a ďalej absolútnu hodnotu Z a Z^2. Absolutnú hodnotu komplexného čísla spočítame použitím Pytagorovej vety, pretože absolutná hodnota je vzdálenost čísla od nuly.

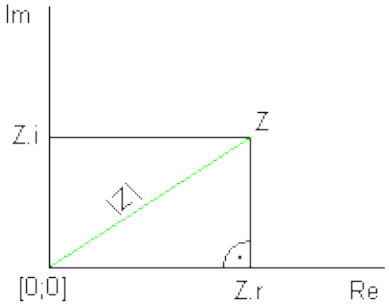

Fig. 8. Výpočet absilútnej hodnoty komplexného čísla

$$Z^2 = Z.r^2 + Z.i^2$$

S výpočtom druhej mocniny Z je to trochu zložitejšie. Z^2 môžeme vyjadriť ako súčin Z * Z, čiže súčin dvoch komplexných čísel, pre ktorý platí všeobecný vzorec:

$$(A + Bi) * (C + Di)$$
$$AC + BCi + ADi + BDi^2, \text{ pričom platí } i^2 = -1$$

Pričom reálna časť násobku je Re = AC – BD a imaginárna časť násobku je Im = BCi + ADi. Pre výpočet je možné určiť:

$$A = Z.r, B = Z.i, C = Z.r, D = Z.i$$

Môžeme preto do predchádzajúceho vzorca dosadiť:

$$Re = Z.r^2 – Z.i^2$$
$$Im = Z.i * Z.r + Z.i * Z.r = 2 * Z.i * Z.r$$

Takže cyklami X a Y prejdeme všetky body množiny. Pre každý bod vynulujeme číslo Z, do C dosadíme [X; Y] a počítadlo iterácií tiež vynulujeme. Spočítame si Z^2 a do Z dosadíme $[Z^2 + C]$. Spočítame si absolútnu hodnotu Z a ak je väčšia ako 2, skončíme a bod zafarbíme príslušnou farbou. Ak nie, zvýšime *iter* o jedna a ideme znovu. Ak vyplytváme všetky pokusy a podmienka sa stále nesplní, zafarbíme bod čierno.

```
For each pixel (Px, Py) on the screen, do:
{
  x0 = scaled x coordinate of pixel (scaled to lie in the Mandelbrot X scale (-2.5, 1))
  y0 = scaled y coordinate of pixel (scaled to lie in the Mandelbrot Y scale (-1, 1))
  x = 0.0
  y = 0.0
  iteration = 0
  max_iteration = 1000
  while (x*x + y*y < 2*2  AND  iteration < max_iteration) {
    xtemp = x*x - y*y + x0
    y = 2*x*y + y0
    x = xtemp
    iteration = iteration + 1
  }
  color = palette[iteration]
  plot(Px, Py, color)
}
```

Fig. 9. Pseudo kód vykreslenia Mandelbrotovej množiny

V. ZHODNOTENIE

V jazyku C sme vytvorili tri verzie programu na výpočet a vykreslenie obrázku mandelbrotovho fraktálu. Dátový typ údajov, s ktorým sme počítali hodnoty pre jednotlivé pixely bol vo všetkých troch verziách rovnaký a síce sa jednalo o 32 bitové číslo s pohyblivou desatinnou čiarkou. Prvá verzia robila výpočet spôsobom SISD a teda všetky výpočty sa vykonávali skalárne. Druhá verzia využívala registre rozšírenia SSE a teda sme dokázali počítať s vektorom 4 štyroch údajov naraz. Tretia verzia využívala rozšírenie AVX a teda sme počítali s vektorom 8 údajov. Tieto tri implementácie sme testovali spôsobom vykreslenia rôznych veľkostí obrázkov Mandelbrotovho fraktálu. Veľkosti obrázkov, na ktorých sme program testovali boli 500x500, 1000x1000, 2000x2000, 3000x3000, 4000x4000 pixelov. Pre každú veľkosť sa obrázok vykreslil 10x a namerané časy jednotlivých vykreslení sa spriemerovali. Tento test sme aplikovali na všetky tri typy implementácii, ktoré sme vytvorili. Zistili sme nasledovné hodnoty:

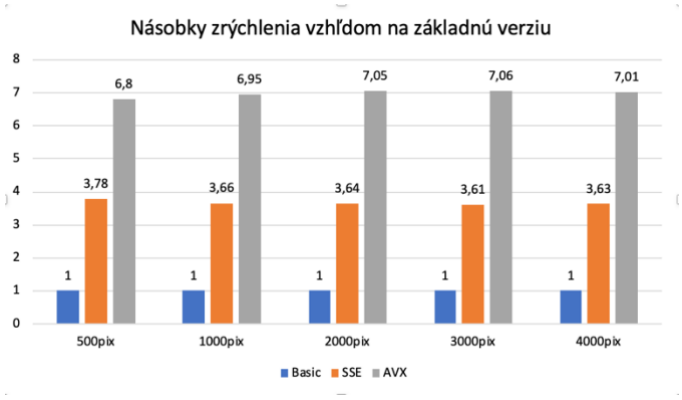

Fig. 10. Tabuľka so zrýchleniami vzhľadom na základnú verziu programu

Zistili sme, že pri všetkých veľkostiach obrázkov, sú jednotlivé zrýchlenia zhruba rovnaké.

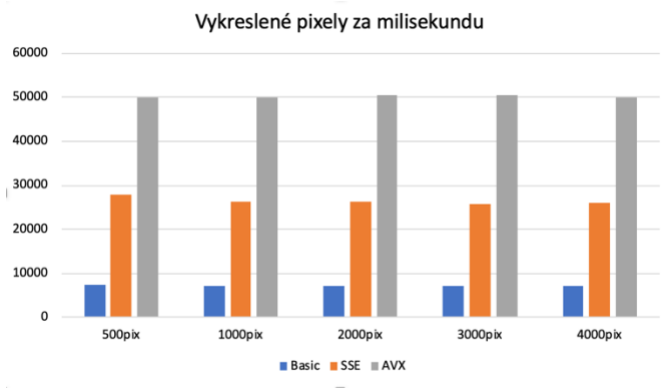

Fig. 11. Tabuľka s počtom vykreslených pixelov za milisekundu

Rovnako aj keď sa pozrieme na počet pixelov vykreslených za milisekundu, je vidno začne zvýšený výkon.

Z nameraných hodnôt je jasne vidieť zrýchlenie pri použití AVX inštrukcií oproti SSE a tiež aj SSE oproti metóde bez špeciálnych inštrukcií. Pri testovaní sme merali len čas potrebný na výpočet obrázku a nemerali sme jeho zapísanie do súboru. Testy prebiehali na procesore Intel Core i5-5257U 2.70 GHz.

VI. Záver

V rámci tejto práce sme vysvetlili rozdiel medzi SISD a SIMD spôsobom spracovania údajov na procesore. Vysvetlili sme rôzne metódy ako funkčnosť SIMD dosiahnuť – či už sa jedná o celkovú novú architektúru alebo doplnenie existujúcej o rozšírenia. Následne sme si vybrali výpočtovú úlohu vykreslenia Mandelbrotovej množiny – čo je náročná úloha na výpočet a zároveň veľmi dobrý kandidát na možné zrýchlenie vďaka SIMD prístupu. Implementovali sme tento algoritmus tromi rôznymi spôsobmi: bez použitia špeciálnych inštrukcií, s použitím SSE a s použitím AVX. Pri testovaní a meraní časov vykreslenia sme zistili výrazné zrýchlenia. Priemerne sa jednalo o 3.6 násobné zrýchlenie pri použití SSE oproti SISD prístupu a 7 násobné zrýchlenie pri použití AVX oproti SISD prístupu.

Autor článku si preto myslí, že využívanie rozšírení procesorov SSE a AVX má zmysel. Podľa trendu, ktorý sme pozorovali a síce, že postupom času vznikajú stále novšie verzie rozšírení, predpokladáme, že aj v budúcnosti sa môžeme dočkať ďalších podobných riešení. Napríklad by mohli pribudnúť registre s dĺžkou 1024 bitov. Alebo môžu vznikať doplnky do inštrukčnej sady, ktoré ešte viac zefektívnia prácu s registrami. Do úvahy tiež pripadá možné vydanie komerčného riešenia, ktoré bude využívať architektúru VLIW ale podmienkou bude aby k danému hardvéru bol dodaný aj veľmi efektívny kompilátor.

Literatúra

[1] J. Fowers et al., "A Configurable Cloud-Scale DNN Processor for Real-Time AI," 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, 2018. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8416814&isnumber=8416802

[2] A. Keliris and M. Maniatakos, "Investigating large integer arithmetic on Intel Xeon Phi SIMD extensions," *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, Santorini, 2014. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6850661&isnumber=6850634

[3] Ch. Lomont, „Introduction to Intel Advanced Vector Extensions“, 2011. URL:https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf

[4] T. Hayes, O. Palomar, O. Unsal, A. Cristal and M. Valero, "Vector Extensions for Decision Support DBMS Acceleration," *2012 45th Annual IEEE/ACM* International *Symposium on Microarchitecture*, Vancouver, BC, 2012. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6493617&isnumber=6493591

[5] Intel Intrinsics Guide. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[6] M. Livesey, F. Costen and X. Yang, "Double precision performance of streaming SIMD extensions instructions for the FDTD computation," Proceedings of the 2012 IEEE International Symposium on Antennas and Propagation, Chicago, IL, 2012. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6348986&isnumber=6347935

[7] W. Hamidouche, M. Raulet and O. Déforges, "4K Real-Time and Parallel Software Video Decoder for Multilayer HEVC Extensions," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 26, no. 1, pp. 169-180, Jan. 2016. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7273890&isnumber=7372356

[8] A. Mondelli, N. Ho, A. Scionti, M. Solinas, A. Portero and R. Giorgi, "Dataflow Support in x86_64 Multicore Architectures through Small Hardware Extensions," *2015 Euromicro Conference on Digital System Design*, Funchal, 2015. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7302318&isnumber=7302233

[9] O. Lempel, A. Peleg and U. Weiser, "Intel's MMX/sup TM/ technology-a new instruction set extension," *Proceedings IEEE COMPCON 97. Digest of Papers*, San Jose, CA, USA, 1997. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=584723&isnumber=12649

[10] P. Hammarlund *et al.*, "Haswell: The Fourth-Generation Intel Core Processor," in *IEEE Micro*, vol. 34, no. 2, pp. 6-20, Mar.-Apr. 2014. URL:http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6762795&isnumber=6786915

[11] J. Kim and S. Cho, "Color multimedia extensions for VLIW architectures," *2007 International Forum on Strategic Technology*, Ulaanbaatar, 2007. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4798658&isnumber=4798500

[12] R. Kong, Y. Sun, X. Wang and L. Bi, "An Image Encryption Algorithm Utilizing Mandelbrot Set," *Chaos-Fractals Theories and Applications, International Workshop on(IWCFTA)*, Kunming, Yunnan China, 2010, URL:doi.ieeecomputersociety.org/10.1109/IWCFTA.2010.70