VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

BAKALÁŘSKÁ PRÁCE

2014 Tomáš Botor

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Porovnání vybraných vývojových platforem z hlediska výkonu v grafických aplikacích
Comparison of Selected Development Platforms in Terms of their Performance in Graphical Applications

2014 Tomáš Botor

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Tomáš Botor**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Porovnaní vybraných vývojových platforem z hlediska výkonu v grafických aplikacích
Comparison of Selected Development Platforms in Terms of their Performance in Graphical Applications

Zásady pro vypracování:

Cílem práce je multiplatformní porovnání výkonu kritických částí aplikací implementovaných v C/C++, CUDA a OpenCL s ohledem na struktury a algoritmy používané v oblasti digitálního zpracování obrazu a počítačové grafiky (manipulace s poli, vektory, maticemi, standardními datovými strukturami apod.). Součástí výsledné práce by měla být sada doporučení pro optimalizaci kódu založená na výsledcích měření kritických částí kódu.

Všechny měřené části kódu implementujte s cílem vytvořit co nejrychlejší kód, a to bez ohledu na čitelnost výsledného kódu. Využijte vhodným způsobem SIMD instrukce (SSEx, AVX) případně knihovny pro paralelizaci kódu (např. OpenMP, TBB). Je-li to možné, v maximální míře využívejte optimalizaci kódu překladačem a zvažte využití intrinsic funkcí a dalších nízkoúrovňových urychlení (prefetch apod.).

1. Implementujte zadanou sadu referenčních struktur a funkcí (bodové operace, konvoluce, násobení a sčítání matic/vektorů, traverzace apod.) v uvedených prostředích.
2. Navrhněte metodiku měření (zhodnoťte možnosti využití tzv. High Performance Counter a porovnejte jej s klasickými způsoby měření času, popište další HW čítače, zhodnoťte vliv cache).
3. Zdokumentujte rozdíly mezi platformami a pokuste se je vysvětlit.

Seznam doporučené odborné literatury:

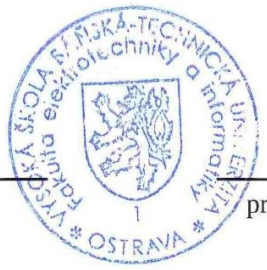
[1] Press, William; Teukolsky, Saul; Vetterling, W.; Flannery, B. Numerical Recipes. 2007. 1262 s. ISBN 978-0-521-88068-8.
[2] Agner Fog. Software optimization resources. Dostupné z WWW: <http://www.agner.org>.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Tomáš Fabián**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2014



doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení:

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Tomáš Bolar

Poděkování:

Děkuji panu Ing. Tomáši Fabiánovi, vedoucímu mé bakalářské práce, za odborné vedení, cenné rady a připomínky, jež mi pomohly zpracovat tuto bakalářskou práci.

Abstrakt:

Cílem této bakalářské práce je vybrat vhodnou metodiku měření času pro kritickou část kódu užívanou v grafických aplikacích. Dále také je zapotřebí vybrat vhodný algoritmus a demonstrovat na něm jednotlivé optimalizační prostředky ať už SSE instrukce či OpenMP direktivy a TBB knihovnu pro paralelizaci kódu. Součástí této práce je i sada doporučení včetně rad a názorných ukázek pro optimalizaci kódu s využitím performance monitor counters a jazyka symbolických adres.

Klíčová slova:

C/C++, performance monitor counters, CUDA, OpenMP, SSE, procesor, cache, instrukce, jazyk symbolických adres, TBB

Abstract:

The aim of this work is to select an appropriate methodology for measuring the time for a critical section of code, used in graphics applications. We also need to select the appropriate algorithm and demonstrate it to the individual optimization techniques such as SSE instructions or OpenMP directives and TBB libraries to parallelize code. Part of this work is a set of recommendations including advice and demonstrations for code optimization using performance monitor counters and assembly language.

Key words:

C/C++, performance monitor counters, CUDA, OpenMP, SSE, processor, cache, instruction, assembly langure, TBB

Seznam obrázků

Seznam tabulek

Seznam výpisů kódů

Obsah

Úvod

Tato bakalářská práce má za cíl analyzovat výkon kritických částí aplikací implementovaných v C/C++ s ohledem na struktury a algoritmy používané v oblasti digitálního zpracování obrazu a počítačové grafiky (manipulace s poli, vektory, maticemi, standardními datovými strukturami apod.).

Proč optimalizovat a profilovat? Optimalizace je důležitý proces, jenž vede ke zvýšení efektivity a ke snížení nároků na výpočetní systém. Pokud nám tedy jde především o co nejlepší využití výpočetního výkonu, je vhodné se o optimalizaci postarat ručně, například pomocí paralelismu s využitím OpenMP direktiv či TBB knihovny.

Díky nástrojům určených k profilování jsme schopni dobře analyzovat kód jako například zjistit, kolik paměti využívá daná funkce, jak moc velký výkon CPU je využit a podobně. Tato místa ve zdrojovém kódu poté můžeme vhodně optimalizovat. Profilování by se dalo rozdělit do dvou kroků. První z nich je měření. V této fázi je dobré změřit celý kód a údaje o těchto výsledcích si zapsat do pomocného souboru. Můžeme například zjistit, kolik mikrooperací, hodinových cyklů, instrukcí či cache miss nastalo, kolikrát se volala daná funkce a podobně. Samozřejmě je také důležité zvolit vhodnou metodiku měření. V druhém kroku jsme poté schopni z naměřených hodnot vytvořit grafy či tabulky. Z těchto statistik se dále snažíme vytipovat místa, kde program stráví nejvíce času a pokoušíme se tato místa vhodně optimalizovat.

Optimalizace se může uskutečňovat na několika úrovních, například optimalizace paměti. V .NET technologii či Javě se o optimalizaci a správu paměti stará Garbage collector. Dále také existuje tzv. low-level optimalizace, kde se při psaní kódu používá jazyk symbolických adres. Ten pak je schopen nejefektivněji optimalizovat danou aplikaci. Programy, s výjimkou na velmi malé, nejsou psány v jazyce symbolických adres z důvodu časové náročnosti. Většina dnes psaných programů je zkompilována z programovacích jazyků vyšší úrovně do jazyka symbolických adres a ručně optimalizována právě až tam. Když není kladen důraz na velikost a efektivitu daných aplikací, jsou tyto aplikace psány v programovacích jazycích vyšší úrovně. Kdysi optimalizaci prováděli samotní programátoři. V dnešní době je zcela běžné nechat optimalizaci na překladači, ovšem mohou nastat situace, kdy je vhodné se postarat o optimalizaci ručně. Také musíme vzít v úvahu fakt, že programu rozumíme daleko lépe než kompilátor. Proto, když nám jde o výrazné urychlení dané aplikace či kusu kódu, je výhodnější spolehnout se sám na sebe a pustit se do ruční optimalizace. Při optimalizaci je nutné si dát pozor na to, že můžeme do zdrojového kódu zanést chyby. Rovněž se jedná o zdlouhavou práci a mnohdy výsledek nemusí odpovídat původnímu záměru.

Pro porozumění a získání představy o tom, co se při optimalizaci zdrojového kódu děje, je vhodné použít výstup do ASM souboru. Samotné zásahy do ASM souboru či programování v jazyce symbolických adres je velmi znalostně a časově náročné. Existují optimalizační prostředky závislé a nezávislé na operačním systému či hardwaru. Pokud nám jde o optimalizaci v nejvyšší možné míře, je nejlepší použít jazyk C/C++ a jako kompilátor je vhodný Microsoft Visual Studio,

jelikož podporuje SSE instrukce, OpenMP direktivy a jsme schopni se přepnout do Disassembly módu či vygenerovat ASM výstup.

Kritické částí kódu jsem měřil na stolním počítači obsahující procesor značky Intel® Core™ i3-540 Processor (4M Cache, 3,06 GHz) se dvěma fyzickými jádry, 32-bitovým operačním systémem a 3,49 GB nainstalované operační paměti. Práci jsem rozčlenil do několika kapitol. První kapitola pojednává o stručné charakterizaci jazyka C/C++, OpenCL a CUDA. Také popisuje jednotlivé optimalizační technologie, jakými jsou například SSE instrukce, TBB a OpenMP. Druhá kapitola definuje měřené veličiny, jakými jsou instrukce, mikrooperace, clock cycle, cache a podobně. Třetí kapitola je věnována analýze měření času, porovnání různých metod jeho měření, využití performance monitor counters, obsahuje rozdíl v měření hodinových cyklů mezi čítači a také návod, jak získat měřené veličiny.

V následující čtvrté kapitole pak bude uveden příklad, na němž bude demonstrována optimalizace. Nejprve bude zobrazen kód bez optimalizačních prostředků s uvedenými parametry, ASM výstupem, naměřeným časem a popisem s komentáři. Poté se jednotlivé části kódu podrobí optimalizaci (paralelismu za pomocí OpenMP direktiv či TBB). Budou zde porovnávány nároky na provedení kódu (ať už počet hodinových cyklů, mikrooperací a instrukcí, tak i čas, který je potřebný k vykonání kódu). Pátá kapitola obsahuje rady a doporučení, jež jsou založeny na výsledcích měření předcházejících kapitol. Poslední kapitola bude věnována závěru a shrne cíly této bakalářské práce.

1. Programovací nástroje

Tato kapitola popisuje základní pojmy, které se budou vyskytovat napříč celou prací. Jedná se o využití programovacího jazyka C/C++, OpenCL a CUDA technologie, jakožto technologie pro paralelizaci výpočtů nad daty, použití TBB knihovny a OpenMP direktiv pro překladač pro paralelizaci výpočtů v C/C++ a dále také SSE instrukce pro vektorizaci kódu. Rovněž jsou zde popsány vnitřní funkce překladače (intrinsic funkce), díky kterým lze rovněž dosáhnout rychlejšího kódu za předpokladu správného nastavení kompilátoru. Součástí této kapitoly je i popis přímého zobrazení strojového kódu či ASM výstupu, který bude v této práci sloužit především pro zobrazení a porovnávání změn při optimalizaci v jazyce symbolických adres. Další podkapitola je věnována disassemblerům, jejichž výstup bude rovněž porovnáván například s ASM výstupem.

1.1 C/C++

Jazyk C je nízkoúrovňový programovací jazyk, mající velmi malou abstrakci nad daty. Poskytuje nízkoúrovňový přístup k operační paměti a je tedy dobrý pro optimalizaci kódu. Většina C++ kompilátorů je schopna generovat výstup jazyka symbolických adres, který je užitečný pro kontrolu, jak dobře kompilátor optimalizuje určitou část kódu. Microsoft Visual Studio je kvalitní kompilátor s mnoha funkcemi. Podporuje 32-bit i 64-bit verzi systému Windows a také můžeme využívat OpenMP direktivy pro paralelizaci kódu[1]. Assembler je program, jenž překládá programy z assembly jazyka do strojového kódu. Assembly language neboli jazyk symbolických adres je nízkoúrovňový programovací jazyk, který je tvořen symbolickou reprezentací jednotlivých strojových instrukcí a konstant potřebných pro vytvoření strojového kódu programu pro daný procesor.

1.2 OpenCL

OpenCL je otevřený standart pro multiplatformní paralelní programování moderních procesorů, jež se nacházejí v osobních počítačích, serverech a dalších zařízeních. OpenCL (Open Computing Language) výrazně zvyšuje rychlost pro široké spektrum aplikací v mnoha tržních kategoriích počínaje herním průmyslem a vědeckým či lékařským softwarem konče[7]. SPIR (Standard Portable Intermediate Representation) mapuje kód z OpenCL C programovacího jazyka do LLVM IR. LLVM (dříve zkratka pro Low Level Virtual Machine) je infrastruktura pro překladač napsaná v C++, navrhnutá pro optimalizaci programů napsaných v libovolném programovacím jazyce[7].

1.3 CUDA

CUDA (Compute Unified Device Architecture) je paralelní výpočetní platforma a programovací model, jenž byl vynalezen společností NVIDIA. Umožňuje strmý nárůst výpočetního výkonu tím, že využívá sílu grafického procesoru (GPU). Parallel Thread Execution (PTX) definuje virtuální stroj a ISA (instruction set architecture) pro všeobecné účely paralelního programování. Programy PTX jsou přeloženy v době instalace na cílový hardware instrukční sady. PTX je navržen tak, aby byl účinný na grafických procesorech společnosti NVIDIA a podporoval výpočetní funkce definované architekturou NVIDIA Tesla. Vysokoúrovňové kompilátory pro jazyky jako je CUDA a C/C++ generují PTX instrukce, které jsou optimalizované a přeloženy do nativních instrukcí cílové architektury[8].

1.4 OpenMP

OpenMP je standard pro paralelní zpracování v jazyce C++ a Fortran. Jedná se o direktivy pro překladač a sadu knihoven. Tento standard je podporován společnostmi Intel, Microsoft a PathScale[1]. Pro využívání OpenMP direktiv je třeba mít přiloženou hlavičku omp.h a dále je nutné nastavit ve vlastnostech projektu menu *C/C++* submenu *Language* vlastnost *OpenMP Support* na hodnotu *Yes*. V jistých případech se také při použití OpenMP doporučuje nastavit přesný počet vláken, jež budou tento program vykonávat. Při krátkém časovém intervalu je vhodné použít vlastnost zvanou afinitu procesorů a nastavit ji na jeden procesor.

1.5 SSE instrukce

Moderní mikroprocesory mají SIMD (Single Instruction Multiple Data) instrukce pro manipulaci s dvěma nebo více částmi stejných dat. Kompilátor je schopen využít SIMD instrukce automaticky v jednoduchých případech, ale člověk programátor je často schopen toto udělat mnohem lépe a to především organizací dat do vektorů, které se vejdou do SIMD registru, a rovněž i organizaci toku celého algoritmu[1]. SSE je rozšíření instrukční sady procesoru o podporu zpracování datových toků formou SIMD architektury, přičemž toto rozšíření spočívá v přidání speciálních 128, 256 nebo 512-bitových registrů (podle verze SSE) a sady instrukcí pro manipulaci s nimi. Tyto registry mohou být rozděleny například na čtyři 32-bitové skaláry, nebo pro 128 bitový XMM registr to může být například i 2× double, 4× float či 16× char. Pokud například vynásobíme dva vektory o čtyřech prvcích, jež každý z vektorů je uložen ve 128-bitovém registru, tak jsou tato 4 násobení zredukována do pouhé jedné operace.

1.6 Vnitřní funkce překladače

Vnitřní funkce překladače (označované taky jako intrinsic funkce) a vektorové třídy jsou velmi doporučovány, jelikož jsou mnohem bezpečnější a jednodušší než použití jazyka symbolických adres, protože se kompilátor stará o přidělování registrů nebo volání konverzí, které je obtížné při psaní v jazyce symbolických adres sledovat. Microsoft, Intel a Gnu C++ kompilátor podporují intrinsic funkce. Většina intrinsic funkcí generuje právě jednu strojovou instrukci. Intrinsic funkce jsou tedy ekvivalentem k assembly instrukcím. Programování s použitím intrinsic funkcí je druh high-level assembly. Lze je snadno kombinovat s C++ jazykovými konstrukcemi jakými například jsou smyčky, if-struktury, funkce, třídy a přetěžování operátorů.

Existence intrinsic funkcí umožňuje provádět snadněji programátorské úlohy, jež kdysi požadovaly programování v syntaxi jazyka symbolických adres. Některé výhody použití intrinsic funkcí jsou následující[1]:

- Není se potřeba učit syntaxi jazyka symbolických adres.
- Bezproblémová integrace do C++ kódu.
- Větve, smyčky, funkce, třídy … atd. Lze snadno provádět s C++ syntaxí.
- Kód je přenosný na téměř všechny x86 platformy: 32-botivý a 64-bitový Windows, Linux, Mac OS, atd.
- Kód je kompatibilní s Microsoft, Gnu a Intel překladači.
- Programátor se nemusí starat o to, který registr je použit pro danou proměnnou.

Mezi nevýhody použití intrinsic funkcí například patří:

- Ne všechny assembly instrukce mají intrinsic ekvivalenty.
- Názvy funkcí jsou někdy dlouhé a obtížně se pamatují.
- Výraz s mnoha intrinsic funkcemi vypadá ošklivě a špatně se čte.
- Kompilátor může upravit kód nebo jej provádět méně účinným způsobem, než programátor zamýšlel.
- Neodborné používání intrinsic funkcí může mít za následek menší efektivitu než jednoduchý C++ kód.

Pokud bychom vzali v úvahu jednoduchý algoritmus a použili v něm metodu memset pro demonstraci intrinsic funkcí, vypadalo by to následovně. Samozřejmě je zapotřebí také povolit intrinsic funkce tak, že ve vlastnostech projektu se dostaneme do menu s názvem *C/C++*, dále na kartě *Optimization* vyplníme pole s názvem *Enable Intrinsic Functions* na hodnotu *Yes(/Oi)*.

```
#pragma intrinsic
void main()
{
   char buffer[] = "This is a test of the memset function";

   printf( "Before: %s\n", buffer );
   memset( buffer, '*', 4 );
   printf( "After:  %s\n", buffer );
}
```

Výpis kódu 1- Ukázka Intrinsic funkce

Při přímém zobrazení strojového kódu (Disassembly window) se pod metodou memset() skrývá následující jediná instrukce:

```
mov         dword ptr [ebp-30h],2A2A2A2Ah
```

Výpis kódu 2 - Intrinsic v Disassembly window

Pokud bychom porovnali původní algoritmus bez #pragma intrinsic, tak se pod metodou memset zobrazí následující pětice instrukcí.

```
push        4
push        2Ah
lea         eax,[ebp-30h]
push        eax
call        @ILT+110(_memset) (9F1073h)
```

Výpis kódu 3 - Disassembly window bez použití Intrinsic funkcí

Při použití intrinsic funkcí je také evidentní urychlení některých výpočetních operací. Například pokud bychom deset čísel datového typu float podrobili metodě abs(), která vypočítává absolutní hodnotu z těchto čísel, došli bychom k zajímavým výsledkům. V mém případě byly výsledné hodnoty následující. Při použití intrinsic funkcí jsem se dostal na hodnotu 0,33 ms, zatímco bez využití intrinsic funkcí se čas vyšplhal až na 2 milisekundy, což je o téměř 84 % větší časový interval.

1.7 Přímé zobrazení strojového kódu

Pomocí Disassembly Window můžeme vidět kód, jemuž odpovídají instrukce vytvořené kompilátorem. Pokud ladíme řízený kód, tyto assembly instrukce odpovídají nativnímu kódu vytvořeného Just-In-Time kompilátorem. Kromě assembly instrukcí může disassembly Windows zobrazit další nepovinné údaje, jakými můžou být například čísla řádků včetně zdrojového kódu, ke kterému jednotlivé assembly instrukce patří, symboly jmen pro adresy paměti, opcode – bajtová reprezentace skutečného stroje nebo MSIL instrukce.

Do módu disassembly window se lze dostat následujícím způsobem. Nejprve je zapotřebí přidat breakpoint do části kódu, která nás zajímá. Pak stačí stisknout klávesu F10 a začít debbugovat. Jakmile se dostaneme k námi umístěnému breakpointu, stačí jednoduše stisknout Ctrl + Alt + D.

1.8 ASM kód

Pokud chceme vygenerovat ASM soubor, musíme ve vlastnostech projektu pod výběrem *C/C++* zvolit nabídku *Output Files* a zde vyplnit políčko pod názvem *Assembler Output* na hodnotu například *Assembly-Only Listing*. V tomto případě se vygeneruje soubor pouze s assembly kódem. Můžeme si zde zvolit i jiné možnosti, například *Assembly with Source code, Assembly with Machine code či Assembly with Machine code and Source.* Dále je zapotřebí pro lepší orientaci

v tomto vygenerovaném ASM souboru se přepnout do *Release* módu a zde opět ve výběru *C/C++* zvolit nabídku *Optimalization* a políčko *Optimalization* nastavit na hodnotu *Disabled*.

1.9 Disassembler

Disassembler slouží ke zpětnému generování strojových kódu z námi vytvořených *.exe* souboru. Disassemblerů existuje řada a některé jsou dostupné i online. Například z webové stránky http://onlinedisassembler.com/odaweb/

1.10 Threading Building Blocks

Threading Building Blocks (TBB) knihovna, firmy Intel, poskytuje vývojářům softwaru řešení pro umožnění paralelismu v C++ aplikacích. Výhodou TBB knihovny je paralelní programování a tím zvýšení výkonu, a škálovatelnost snadno přístupná pro softwarové vývojáře, jež používají smyčky a píší aplikace založené na úkolech. Knihovna obsahuje řadu obecných paralelních algoritmů, podporu závislosti a toku dat grafů, lokální úložiště pro práci s vlákny, plánovač úloh pro programování založené na úkolech, synchronizační nástroje, škálovatelné paměti alokátoru a podobně. [14]

TBB knihovna je dostupná ke stažení na webových stránkách firmy Intel. Po stažení je zapotřebí v Microsoft Visual Studiu importovat stažené soubory a složky (*include* a *lib/ia32*) ve vlastnostech projektu v kartách menu *C/C++* a *Linker*. Dále je nutné, aby byla v počítači přítomna knihovna *tbb_debug.dll* popřípadě *tbb.dll*. Je možné je stáhnout z internetu. Poté už je jen stačí umístit do zdrojové složky řešení projektu. Samotné řešení je pak doporučené spouštět v Microsoft Visual Studiu stiskem kláves C*trl + F5*.

1.11 Rozdíly mezi OpenMP a TBB

Jak TBB, tak OpenMP představují multiplatformní nástroje na paralelizaci kódu. Oba dva optimalizační prostředky vytvářejí a spravují pool vláken, rovněž se starají o synchronizaci a plánování úkolů. OpenMP je poměrně jednodušší na realizaci paralelismu a to především smyček. Například před smyčku, jež chceme paralelizovat, stačí připsat *#pragma omp parallel for* a kompilátor si pak se vším poradí sám. Samozřejmě je důležité mít povolenou OpenMP podporu v nastavení. Nevýhodou OpenMP je tedy fakt, že musí být podporována kompilátorem. Při optimalizaci kódu pomocí TBB je nutné stáhnout požadované soubory a postarat se o jejich importování do vývojového prostředí. Výhodou je, že použití TBB knihovny jsou nezávislé na překladači. Nevýhodou TBB je fakt, že programátor musí strávit více času ke studiu knihovny. Například pro paralelizaci smyčky (například násobení matic) je nutné vytvořit speciální třídu pro toto použití. Dochází k daleko většímu nárůstu zdrojového kódu, než v případě OpenMP direktiv.

OpenMP je pro uživatele přívětivější technika paralelizace kódu pro její jednoduchost a časovou nenáročnost pro naučení. TBB knihovna zato nabízí komplexnější paralelizaci algoritmů či škálovatelnou paměť alokátorů.

Pro lepší představu v psaní optimalizovaného kódu jsem se rozhodl demonstrovat výše uvedené techniky na algoritmu násobení matic. Pro OpenMP direktivy stačí pouze před smyčku, jež chceme paralelizovat, napsat *#pragma omp parallel for*., jak je znázorněno ve Výpisu kódu 4.

```
void MatrixMultiply (int matrix1[row][column], int matrix2[row][column],
int matrix[row][column])
{
#pragma omp parallel for
for (int i =0;i<row;i++)
      for (int y=0;y<column;y++)
            for (int z=0;z<row;z++)
            {
                  matrix[i][y] += matrix1[i][z] * matrix2[z][y];
            }

}
```

Výpis kódu 4 - Ukázka OpenMP optimalizace

Naproti tomu ve Výpisu kódu 5 vidíme optimalizaci pomocí TBB knihovny. Je zapotřebí vytvořit třídu podle daného předpisu a následně volat tuto optimalizaci z dané metody následovně: *parallel_for(blocked_range<int>(0,size), Multiply());*

```
class Multiply
{
public:
    void operator()(blocked_range<int> r) const {
        for (int i = r.begin(); i != r.end(); ++i) {
            for (int j = 0; j < size; ++j) {
                for (int k = 0; k < size; ++k) {
                    arrayC[i][j] += arrayA[i][k] * arrayB[k][j];
                }
            }
        }
    }
};
```

Výpis kódu 5 - Ukázka TBB optimalizace

2. Definice měřených hodnot

Tato kapitola má za úkol popsat naměřené hodnoty, jakými jsou například: cache miss, clock cycle, instrukce a mikrooperace. Jak získat tyto hodnoty je popsáno v podkapitole 3.3 a demonstrováno na příkladech v kapitole 4. Rovněž tato kapitola popisuje některé problémy, jež s těmito měřenými hodnotami souvisí, například u cache jde o cache miss či s pipeliningem souvisí problém a to predikci větvení včetně techniky odstranění tohoto problému a tou je technika rozbalování smyček. Také je zde definován procesor a registr procesoru s příkladem ukládání parametrů funkce do registru. Součástí je i charakteristika CPU-bound a Memory-bound systémů a jejich porovnání.

2.1 Cache a clock cycle

Vyrovnávací paměť (Cache Memory) je velmi rychlá paměť, která je zpravidla umístěna mezi procesorem a hlavní pamětí výpočetního systému. Ve vyrovnávací paměti je uložena ta část hlavní paměti, která je právě procesorem používána. Vyrovnávací paměť může být také umístěna mezi hlavní paměť a velkokapacitní vnější paměť. Paměť cache mí omezenou kapacitu. Jsou v ní uloženy kopie dat z hlavní paměti výpočetního systému. Nacházejí-li se požadované údaje, data nebo instrukce v paměti cache, jsou přečteny z této rychlé paměti (cache hit) a není uskutečněn přístup do relativně pomalé hlavní paměti (cache miss) [3].

Jakmile se vyrovnávací paměť naplní, je nutno rozhodnout o tom, který blok vyrovnávací paměti má uvolnit místo pro nový blok z hlavní paměti. Nejrozšířenějším algoritmem je algoritmus LRU (Least Recently Used = nejdéle nepoužívaný). Vyřazuje se vždy ten blok, který byl nejdéle procesorem nepoužit[3].

Cache se dělí do několika úrovní (levelů). První úroveň je nejrychlejší, ale má jen velmi malou kapacitu. Čím vyšší úroveň, tím klesá rychlost, ale zvyšuje se paměť. Typicky se úroveň 1 (dále již L1) data cache pohybuje v rozmezí od 8 do 64 KB a L2 cache od 256 KB do 2 megabajtů. Ovšem můžeme se dnes již setkat i s L3 cache. Pokud celková velikost všech dat v programu je větší než druhá úroveň vyrovnávací paměti a data jsou uloženy po celé paměti, nebo se k nim přistupuje nesekvenčním způsobem, je pravděpodobné, že přistup do paměti je patrně největším spotřebitelem času v daném programu. Čtení nebo zápis do proměnné v paměti zabere 2 – 3 hodinové cykly, pokud je v cache. Pokud není v cache, zabere tento proces až několik set hodinových cyklů [1].

Jelikož mají dnes počítače různou rychlost, je vhodnější určovat dobu potřebnou k vykonání jisté operace v hodinových cyklech (clock cycle). Například pokud triviální operace nad dvěma čísly datového typu float trvá 5 hodinových cyklů, tak tato operace bude stále trvat pět hodinových cyklů, i když se taktovací frekvence procesoru zdvojnásobí.

2.2 Mikrooperace a strojová instrukce

Mikrooperace je činnost procesoru nebo jeho logicky ohraničené části během jedné fáze. Například: přenos dat mezi registry[3]. Strojová instrukce je kódovaný příkaz k vykonání strojové operace. Úplný soubor strojových instrukcí tvoří tzv. strojový jazyk nebo strojový kód. Programování ve strojovém kódu je namáhavé a vzniklé programy jsou nepřehledné. Proto se častěji programuje v jazyce symbolických adres (assembly language)[3]. Instrukce je možné chápat jako posloupnost mikrooperací. Například instrukce read-modify může být rozdělena na mikrooperaci read a mikrooperaci modify [1].

Pro představu vezměme v úvahu následující příklady. Pokud máme jednoduchou rovnici *a=2+3*, výsledek bude roven právě jedné instrukci. Jestliže ovšem rovnici pozměníme na následující: *a=b+3*, bude zapotřebí již tři instrukce k provedení této operace. Když manipulujeme s prvky v poli, tak rovnice: *pole[i]+=pole[i] + 2* nás bude stát sedm instrukcí.

2.3 Registr procesoru

Omezený počet proměnných může být uložen v registru procesoru místo v hlavní paměti. Registr je malý kousek paměti uvnitř CPU pro dočasné ukládání dat. Proměnné, které jsou uloženy v registru, jsou dostupné velmi rychle. Všechny optimalizační překladače automaticky vybírají nejčastěji využívané proměnné ve funkcích a ukládají je do registrů. Počet registrů je velmi omezený. V 32-bitovém operačním systému je přibližně 6 celočíselných registrů a v 64-bitovém operačním systému jich je 14. Proměnné datového typu float používají jiný druh registrů. K dispozici je 8 registrů v 32-bitovém operačním systému a 16 v 64-bitovém[1].

Pokud bychom vzali v úvahu 64 bitovou verzi softwarové konvence při předávání parametrů, výsledek by byl pro jednotlivé datové typy následující. U datového typu float a double dochází k předávání prvních 4 parametrů funkce do registrů XMM0 – XMM3, další se ukládají do zásobníku. U datového typu integer se první 4 parametry ukládají do registrů RCX, RDX, R8 a R9, ostatní se ukládají na zásobník[10].

Například: funkce1(int a, double b, int c, float d); //a je uloženo v registru RCX, b je v registru XMM1, c se nachází v registru R8 a v registru XMM3 je hodnota d[10].

2.4 Procesor

Procesorem se rozumí základní jednotka počítače, tj. logický automat pro zpracovávání informací, obsahující hlavně aritmetickou jednotku a řadič. „Počítač bez periferních zařízení a bez hlavní paměti“[3]. Mikroprocesor má univerzální strukturu seskupenou kolem jediného výkonného členu. Jeho instrukční soubor je koncipován především na výpočty a logické funkce. Je přednostně orientován na operace nad slovy. Slovo (word) je skupina několika slabik, které se v počítači zpracovává jako celek. Slabika (byte) je skupina obvykle osmi bitů[3].

2.5 FPU

Floating point unit bývá označován jako matematický koprocesor. Jedná se o jednotku, která realizuje instrukce pohyblivé řádové čárky.

2.6 CPU-bound, Memory-bound, Cache-bound a I/O-bound

Pravděpodobně nejrozšířenější skupinou jsou CPU-bound systémy. CPU bound znamená, že rychlost, kterou je program vykonáván, je limitována rychlostí CPU. Například to může být násobení malých matic. Na rozdíl od jiných systémů, ve kterých počítač spoléhá na komponenty, jež pomáhají při zpracovávání, v CPU-bound systémech spoléhá počítač pouze na CPU. Když chce uživatel urychlit počítač, obvykle stačí přidání více paměti RAM nebo jiných komponent. V CPU bound systémech by měl být vylepšen pouze procesor, jelikož na těchto dalších paměťových rozšířeních nezáleží.

I/O bound systémy mají tu vlastnost, že rychlost, která je zapotřebí k vykonání programu, je omezena rychlostí I/O subsystému. Například se může jednat o program, jenž počítá řádky ve velkém souboru. U memory bound je limitujícím faktorem řešení daného problému rychlost přístupu do paměti. Cache bound znamená, že rychlost programu je omezena rychlostí a množstvím cache paměti, jež je k dispozici. I/O bound systémy jsou pomalejší než memory bound systémy. Ty jsou pomalejší než cache bound systémy. A cache bound systémy by měly být pomalejší než CPU bound systémy.

Výhody CPU bound systému jsou stejně početné, jako jeho nevýhody. CPU-bound systémy jsou nejrychlejší a bývají také často nejpoužívanější. Pokud jde o programy s výpočty, bude CPU bound systém nejvhodnější. Namísto toho, aby uživatel vylepšoval další komponenty, bude stačit, aby se soustředil pouze na modernizaci procesoru pro zlepšení celkového výkonu[9].

2.7 Zřetězené zpracování instrukcí, predikce skoku a větvení

Vysoké rychlosti moderních mikroprocesorů se dosahuje pomocí zřetězeného zpracování instrukcí, kde jsou instrukce načítány a dekódovány ještě předtím, než jsou provedeny. Avšak struktura zřetězeného zpracování instrukcí má jeden problém. Jakmile se narazí v kódu na if-else strukturu, mikroprocesor neví, kterou z těchto větví má zřetězit. V případě, že procesor zřetězuje nesprávnou větev, tak se chyby zjistí až po 20 hodinových cyklech a veškerá práce, která byla provedena do této chvíle (například dekódování a načítání instrukcí) byla naprosto zbytečná. Mikroprocesoroví designéři se snažili tento problém vyřešit. Nejdůležitější metoda, která se používá, je predikce skoku. Moderní mikroprocesory používají pokročilé algoritmy k tomu, aby se pokusily správně předpovědět, kterou větví se bude program ubírat.[1]

Lokální branch predictor má oddělený buffer, do kterého ukládá historii pro každý podmíněný skok. Globální branch predictor nevede oddělenou historii záznamů pro každý skok. Namísto toho vede společnou historii všech podmíněných skoků. Výhodou sdílené historie je, že každý vztah mezi různými podmíněnými skoky je součástí tvorby předpovědí. [4]

Saturating counter nebo bimodal predictor je konečný automat se čtyřmi stavy: *Strongly not taken, Weakly not taken, Weakly taken, Strongly taken*. Když se vyhodnocuje větev, je odpovídající stav stroje aktualizován. Větve vyhodnocené jako *not taken* dekrementují stav k *Strongly not taken* a větve vyhodnocené jako *taken* zvyšují stav k *Strongly taken*. Výhodou dvoubitového čítače v jednobitovém schématu je, že podmíněný skok se musí lišit dvakrát od toho, co se provedlo nejvíce v minulosti před predikcí změn. [4]

Statická predikce skoku je nejjednodušší technika predikcí skoků, protože nespoléhá na informace o dynamické historii vykonávajícího se kódu. Místo toho se předpovídá výsledek větve založený výhradně na branch instrukcích. Dynamická predikce skoku zpracovává tabulku historie skoků. Do té se ukládá informace o tom, zda se skákalo či nikoliv, když se instrukce vykonala naposled. Každá doposud vykonaná instrukce má v tabulce jeden záznam. Existují čtyři způsoby odstraňování větví [6]:

- Uspořádat kód na základní souvislé bloky
- Unroll loops – rozbalování smyček
- Použít CMOV instrukce
- Použít SETCC instrukce

Jako názorný příklad pro analýzu predikce větvení jsem zvolil následující triviální strukturu. Výpis kódu 6:

```
int a=1;
int b, c;

if(a!=1)
{
b = a;
}
else
{
c=a;
}
```

Výpis kódu 6 – Vzor IF struktury

Z výše uvedeného výstupu Výpis kódu 6 je patrné, že při porovnání na řádku *if (a!=1)* dojde k nesplnění podmínky, program skočí do větve *else* a pokračuje dále ve vykonávání instrukcí. Pokud by operand *a* nebyl roven číslu 1, program by se vykonával dál a jakmile by narazil na identifikátor *else* , následující instrukcí *jmp* by tuto větev přeskočil a pokračoval by dále.

V tomto případě uvažuji, že porovnávaný operand *a* je roven hodnotě 1. Když jsem podmínku *if (a==1)* vykonával v cyklu tisíckrát za sebou, dostal jsem se k průměrné hodnotě 1,945 mikrosekund. Pokud jsem ovšem negoval podmínku na *if (a!=1),* průměrná hodnota se vyšplhala

až na 2,35 mikrosekund, což je přibližně o 21 % delší vykonávání podmínky než v opačném případě.

Výsledky z programu od Agnera Foga pro podmínku *if (a==1)* jsou následující: počet mikrooperací je roven 7, instrukcí se provedlo 5 a hodinových cyklů proběhlo také 5. Pokud podmínku negujeme *if (a!=1),* dostaneme tyto hodnoty: mikrooperací se vykonalo 6, instrukce proběhly 4 a rovněž i hodinové cykly jsou rovny číslu 4.

2.8 Rozbalování smyček

Kritická část programu je téměř vždy smyčka. Hodinová frekvence moderních počítačů je tak vysoká, že i časově nejnáročnější instrukce, cache miss a neefektivní výjimky jsou vykonány během zlomku mikrosekundy. Zpoždění způsobené neefektivním kódem je tedy patrné pouze při velkém množství opakování. U smyček je instrukce potřebná pro skok zpět na začátek smyčky a má za úkol rozhodnout, kdy je smyčku třeba ukončit. Optimalizace právě těchto instrukcí je poměrně obecná technika, která může být použita v mnoha situacích. Rozvíjení smyček (anglicky označováno jako unroll loops) je způsob optimalizace kódu, kdy se počet celkových iterací smyček snižuje.[1] Vezměme v úvahu následující obecný příklad.

Ve výpisu kódu 7 – Rozvíjení smyček je v prvním sloupci uveden běžný cyklus a v pravém sloupci je tentýž cyklus již modifikován technikou rozvíjení smyček. V důsledku této změny se provádí pouze deset opakování namísto padesáti. Je tedy zapotřebí provést pouze deset procent skoků, což vede ke snížení režie smyčky. Na druhou stranu je třeba dodat, že toto manuální rozvíjení smyček způsobilo nárůst zdrojového kódu ze tří na sedm řádků, které musí být kontrolovány, laděny a kompilátor musí přidělit více registrů pro ukládání proměnných. Může zde také dojít k problémům při pozdějších optimalizacích, například v tom, že celkový počet iterací nebude dělitelný v tomto případě pěti.

```
int i;
for ( i=0; i<50; i++){
      foo(x);
}
```

```
int i;
for ( i=0; i<50; i+=5){
      foo(x);
      foo(x+1);
      foo(x+2);
      foo(x+3);
      foo(x+4);
}
```

Výpis kódu 7 - Rozvíjení smyček

Předcházející teorii jsem se rozhodl demonstrovat na následujícím příkladu. Vzal jsem v úvahu předchozí cyklus, ale namísto sta iterací jsem zvolil číslo 120, pro jeho větší dělitelnost a tím i patrnějším rozdílům mezi jednotlivými stupni rozbalování. Funkci *foo()* jsem nahradil jednoduchou funkcí, jež má za úkol sečíst dvě čísla. Následující tabulka zobrazuje výsledky, které jsem naměřil jak pomocí funkce *QueryPerformanceCounter()*, tak i pomocí programu od pana

Agnera Foga (postup měření je uvedený v podkapitole 3.3). Tabulka 1 – Manuální rozbalování smyček je rozdělená do šesti sloupců. Jednotlivé řádky znázorňují daný počet iterací. Druhý sloupec *čas (µs)* značí čas naměřený funkcí *QueryPerformanceCounter()*, jež je vybrána v podkapitole 3.1 jako funkce pro měření času v jazyce C/C++ s nejmenší odchylkou měření. Sloupec *Clock cycles* obsahuje počet hodinových cyklů, sloupec *Instructs* zase počet instrukcí a *µOps* zase sumu vykonaných mikrooperací. Poslední sloupec tabulky je věnován cache missům. První řádek značí žádné rozbalení a poslední řádek je maximální rozbalení smyčky.

Tabulka 1 - Manuální rozbalování smyček

Manuální rozbalování smyček					
Počet iterací	**čas (µs)**	**Clock cycles**	**Instructs**	**µOps**	**Cache miss**
120	3,38	9 758	3 364	22 806	0
60	3,32	9 604	3 004	22 326	0
40	3,30	9 564	2 884	22 166	0
30	3,27	9 544	2 824	22 086	0
24	3,29	9 534	2 788	22 038	0
20	3,33	9 524	2 764	22 006	0
15	3,30	9 514	2 734	21 966	0
12	3,29	9 510	2 716	21 942	0
10	3,31	9 504	2 704	21 926	0
8	3,33	9 500	2 692	21 910	0
6	3,33	9 496	2 680	21 894	0
5	3,40	9 494	2 674	21 886	0
4	3,35	9 494	2 668	21 878	0
3	3,39	9 492	2 662	21 870	0
2	3,43	9 490	2 656	21 862	0
1	3,55	9 488	2 650	21 854	0

Při použití inline funkce pro součet se výsledky neliší od výše uvedené tabulky.

2.9 Předávání parametrů

X64 architektura poskytuje 16 univerzálních registrů (celočíselné registry), jakož i 16 XMM registrů, jež jsou k dispozici pro použití s plovoucí desetinnou čárkou. Registry RBX, RBP, RDI, RSI, RSP, R12, R13, R14, R15 a jsou považovány za nonvolatile a registry RAX, RCX, RDX, R8, R9, R10, R11 jsou považovány za volatile [10].

Následující tabulka popisuje, jak se každý registr používá při volání funkcí:

Tabulka 2 - Přehled registrů a jejich užití u Windows x64

Registr	Status	Použití
RAX	Volatile	Vrací hodnotu registru
RCX	Volatile	První argument datového typu integer
RDX	Volatile	Druhý argument datového typu integer
R8	Volatile	Třetí argument datového typu integer
R9	Volatile	Čtvrtý argument datového typu integer
R10:R11	Volatile	Musí být zachován podle potřeby volajícím, použití u syscall/sysret instrukcí.
R12:R15	Non-volatile	Musí být zachován volaným
RDI	Non-volatile	Musí být zachován volaným
RSI	Non-volatile	Musí být zachován volaným
RBX	Non-volatile	Musí být zachován volaným
RBP	Non-volatile	Může být použit jako *frame pointer*; musí být zachován volaným
RSP	Non-volatile	Ukazatel zásobníku
XMM0	Volatile	První argument datového typu float
XMM1	Volatile	Druhý argument datového typu float
XMM2	Volatile	Třetí argument datového typu float
XMM3	Volatile	Čtvrtý argument datového typu float
XMM4:XMM5	Volatile	Musí být zachován podle potřeby volajícím
XMM6:XMM15	Non-volatile	Musí být zachován podle potřeby volaným

V níže uvedené tabulce 3 je možné se podívat na rozdíly mezi jednotlivými operačními systémy a jejich verzemi. Volatile registry jsou registry, jež slouží k dočasnému uložení hodnot (bez omezení). Také se nazývají scratch registry. Non-volatile registry jsou takové registry, které musí být uloženy před jejich použitím a obnoveny po jejich používání (nazývané jako callee-save registry). Poslední řádek níže uvedené tabulky reprezentuje registry, které slouží pro navrácení požadované typové hodnoty [1].

Tabulka 3 - Porovnání registrů napříč platformami

	32 bit Windows	**32 bit Linux, Mac OS**	**64 bit Windows**	**64 bit Linux, Mac OS**
Volatile registry	EAX, ECX, EDX, ST(0)-ST(7), XMM0-XMM7, YMM0-YMM7, ZMM0-ZMM7, K0-K7	EAX, ECX, EDX, ST(0)-ST(7), XMM0-XMM7, YMM0-YMM7, ZMM0-ZMM7, K0-K7	RAX, RCX, RDX, R8-R11, ST(0)-ST(7), K0-K7, XMM0-XMM5, Všechny YMM/ZMM registry s výjimkou nižších 128 bitů XMM6-XMM15	RAX, RCX, RDX, RSI, RDI, R8-R11, ST(0)-ST(7) K0-K7, XMM0-XMM15, YMM0-YMM15 ZMM0-ZMM31
Non-volatile registry	EBX, ESI, EDI, EBP	EBX, ESI, EDI, EBP	RBX, RSI, RDI, RBP, R12-R15, XMM6-XMM15	RBX, RBP, R12-R15
Registry sloužící k návratu hodnot	EAX, EDX, ST(0), XMM0, YMM0, ZMM0	EAX, ST(0), XMM0, YMM0, ZMM0	RAX, ST(0) XMM0, YMM0, ZMM0	RAX, RDX, ST(0), XMM0, YMM0, ZMM0

V následujícím výpisu kódu je uvedený ASM výstup s popisem registrů i instrukcí. Jedná se o funkci, která postupně sčítá tři čísla, předaných jako parametry funkce, různých datových typů a ukládá je do nově deklarované proměnné. Tu pak celá funkce vrací. Platformou je zde 32 bitová verze systému Windows. Z výše uvedené tabulky si lze povšimnout, že jsou plně využity Calle-save registry, jež jsou nejprve uloženy do zásobníku a na konci procedury zase uvolněny.

```
;     COMDAT ?function@@YANHMN@Z
_TEXT SEGMENT
_r$ = -12                               ; size = 8
_i$ = 8                                       ; size = 4
_f$ = 12                                      ; size = 4
_d$ = 16                                ; size = 8
?function@@YANHMN@Z PROC      ; function, COMDAT
; 5    : {
      push  ebp                         Prolog assembly funkce, předešlý kontext ukládá
      mov   ebp, esp                    do zásobníku. EBP registr je vrchol zásobníku
      sub   esp, 208                    Alokuje místo pro ukládání lokálních proměnných
      push  ebx                         Vložení registru EBX do zásobníku
      push  esi                         Vložení registru ESI do zásobníku
      push  edi                         Vložení registru EDI do zásobníku
      lea   edi, DWORD PTR [ebp-        Načtení efektivní adresy do registru EDI
208]
; 6    : double r;
; 7    : r=i+f;
      fild  DWORD PTR _i$[ebp]          Načte hodnotu datového typu int
      fadd  DWORD PTR _f$[ebp]          Sčítání pro datový typ float
      fstp  QWORD PTR _r$[ebp]          Uložení datového typu float
; 8    : r=d+r;
      fld   QWORD PTR _d$[ebp]          Načte datový typ float
      fadd  QWORD PTR _r$[ebp]          Sčítání pro datový typ float
      fstp  QWORD PTR _r$[ebp]          Uložení datového typu float
; 9    : return r;
      fld   QWORD PTR _r$[ebp]          Načte datový typ float
; 10   : }
      pop   edi                         Odebrání registru EDI ze zásobníku
      pop   esi                         Odebrání registru ESI ze zásobníku
      pop   ebx                         Odebrání registru EBX ze zásobníku
      mov   esp, ebp                    Kopíruje hodnotu z ESP do EBP registru
      pop   ebp                         Návrat z procedury. Epilog assembly funkce,
      ret   0                           předchozí kontext může být obnoven.
?function@@YANHMN@Z ENDP
_TEXT  ENDS
```

Výpis kódu 8 - ASM výstup s popisem registrů a instrukcí

2.10 Spřažení procesorů

Ve správci úloh můžeme nastavit vlastnost zvanou spřažení procesorů (v angličtině processor affinity). Tato vlastnost umožňuje na vícejádrových procesorech využít výkon jejich jader podle potřeby. Pokud klikneme pravým tlačítkem myši na námi vybraný proces a vybereme z následného menu položku spřažení procesorů, vytvoří se nové okno, jež nám umožňuje nastavit, jaké procesory mohou spouštět daný proces. [13] Když nastavíme afinitu na jedno jádro pro delší měřené úseky, dojde ke zvýšení výpočetního výkonu. Nedojde poté k migraci dat mezi jednotlivými jádry a tím i tolika cache missům. Pro nastavování spřažení procesorů je možné využít i řešení přímo ve zdrojovém kódu. V operačních systémech Windows se k tomuto účelu využívá funkce SetProcessAffinityMask.[1] Ta umožní uzamknout daný proces v jednom námi zvoleném jádře

procesoru. Následující výpis kódu znázorňuje, jak by mohlo vypadat uzamčení vykonávaného procesu do jednoho jádra.

```
#include <Windows.h>
#include <iostream>
using namespace std;


int main ()
{
	HANDLE process = GetCurrentProcess();
	DWORD_PTR processAffinityMask = 1; // číslo CPU

	BOOL success = SetProcessAffinityMask(process, processAffinityMask);
	cout << success << endl;


       //algoritmus pro měření

       return 0;
}
```

Výpis kódu 9 - Nastavení afinity

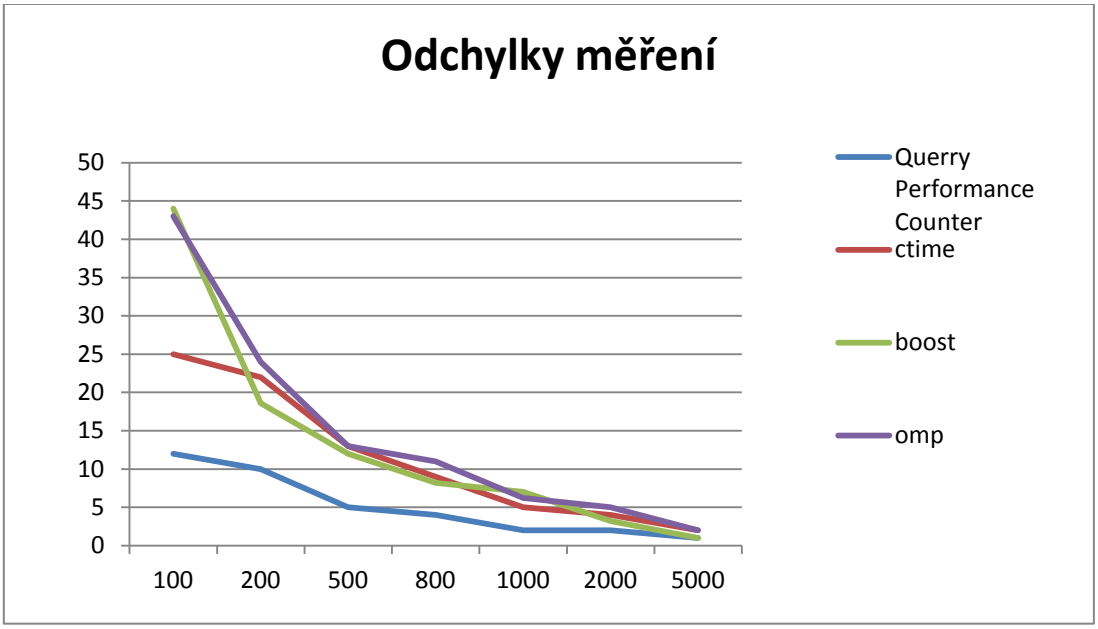
3. Analýza měření času

Tato kapitola je věnována analýze měření času v C/C++, porovnání různých metod jeho měření a popis postupu při měření s využitím programu od pana Agnera Foga, jež využívá performance monitor counters, díky kterým jsme schopni získat informace o počtu provedených instrukcí, mikrooperací, hodinových cyklech a podobně. Tento program od Agnera Foga se bude používat i v následující kapitole pro odpovídající algoritmy. Na základě výsledků měření v podkapitole 3.1 se jedna z vybraných metod (ta s nejmenší odchylkou) použije pro měření klíčových částí kódu v kapitole čtvrté. Podkapitola 3.3 obsahuje vše, co je potřeba udělat k zisku výsledků pomocí performance monitor counters a obsahuje i screenshot z výstupu měřícího programu. Dále je zde také popsán Time-Stamp counter a je zde i znázorněn rozdíl v měření hodinových cyklů mezi výše uvedeným čítačem a programem od pana Agnera Foga.

3.1 Měření času

V jazyce C/C++ existuje mnoho způsobů jak změřit čas určité části kódu. Nabízí se například knihovna *Ctime.h* a využití funkce *clock*, jež měří čas s přesností na jednotky sekund. Pro měření s nejvyšším možným rozlišením se doporučuje volat metodu *QueryPerformanceCounter*, která je schopná zobrazit čas s přesností na mikrosekundy.

Jelikož moderní procesory mění frekvenci v závislosti na zatížení, počet CPU tiků, není možné přímo převést na časové jednotky. Funkce *getCPUTickCount* je rovněž vhodná pro měření času vykonávání metody, protože vrací aktuální počet CPU tiků. V následujícím grafu jsou uvedeny i některé další možné funkce pro měření uplynulého času určité části kódu.



Obrázek 1 – Odchylky měření různými metodami v procentech

Jako algoritmus pro měření části kódu jsem zvolil funkci, jež měla za úkol cyklický výpis celých čísel do konzole. Vždy jsem nejprve inicializoval počáteční hodnotu měření. Následně jsem volal svou funkci a vzápětí metodu, která mi vrátila hodnotu čítače. Vypočítal jsem rozdíl mezi těmito dvěma časovými hodnotami a získal jsem dobu, během které se jistá funkce vykonala. Tuto hodnotu jsem si uložil do pole a celý postup se stejnou metodou několikrát zopakoval. Tím jsem získal pole s několika hodnotami. Poté jsem vypočetl průměr a maximální prvek z pole mých naměřených hodnot. V grafu, který je uveden výše, vidíte procentuální odchylku mezi naměřenou průměrnou hodnotou a jejím maximem v daném poli. Svislá osa značí velikost odchylky v procentech a vodorovná osa zase počet celých čísel, které má funkce vypisovala do konzole. Je tedy zřejmé, že pro měření krátkých časových intervalů je nejefektivnější použít metodu *QueryPerformanceCounter*.

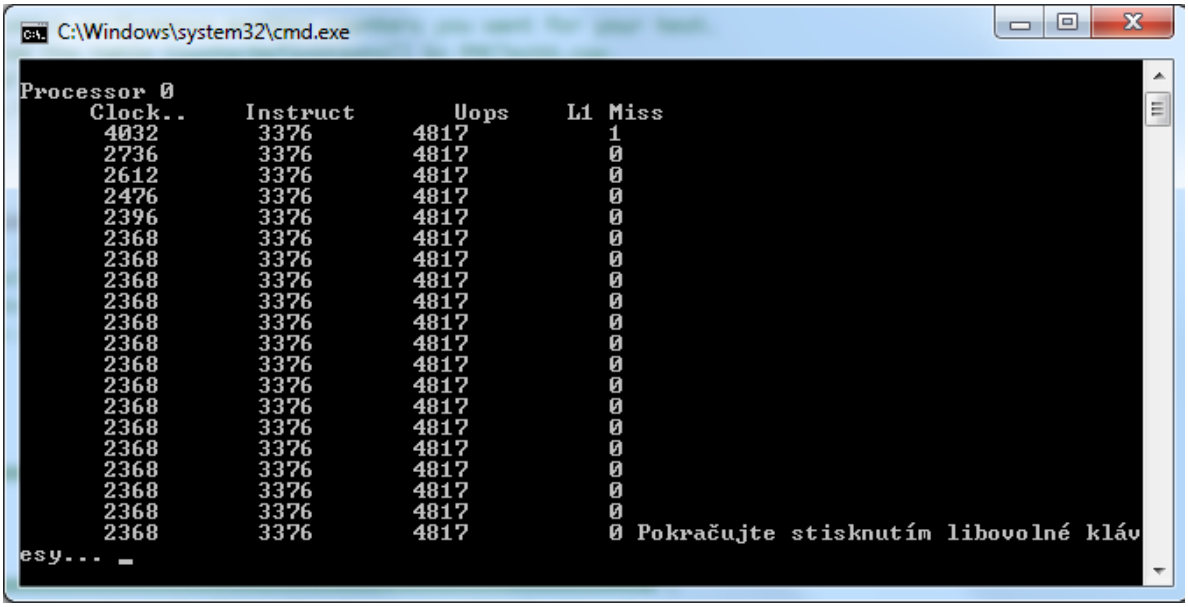
3.2 Performance Monitor Counters

Performance monitor counters je cenný nástroj pro měření výkonu programu (nebo aplikace, služby či ovladače), který může být analyzován pro určení různých překážek v programu. Tyto čítače jsou přítomny ve většině moderních procesorů včetně Intel Pentium, Pentium Pro, P6, Pentium 4, AMD, Cyrix apod. Tyto čítače jsou hardwarové registry spojené s procesorem, měřící různé programovatelné události vyskytující se v procesoru, jakými jsou například: počet instrukcí a mikrooperací, data cache miss či clock cycles. [2]

3.3 Získání výsledků z čítačů

Na webových stránkách pana Agnera Foga je možné stáhnout testovací program pro měření hodinových cyklů či jiných performance monitor counters, jakými mohou být cache misses, branch mispredictions, mikrooperace, instrukce a podobně. Program podporuje jak platformu Windows, tak i Linux včetně 32 a 64 bitových módů. Tento program je určený pro měření kritických částí kódu především pro malé programy. Není doporučen pro profilování celého programu. Po stažení programu z webových stránek je zapotřebí vyextrahovat námi stažený archiv. Po vyextrahování je možné nalézt soubor *textp.txt* který obsahuje detailní popis staženého programu, návod i odpovědi na často kladené dotazy. V mém případě stačilo poté vyextrahovat projekt *vs2010_project_files.zip* a spustit řešení c32 jako správce. Pak už jednoduše je možné vložit část kódu pro změření na místo, jež je označeno jako *### Test code here ###* nebo *### Test code start ###* v souboru *PMCTestB*.

Je možné si vybrat mezi různými PMC čítači vložením požadovaného *counter id* na *CounterTypesDesired* v souboru *PMCTestB*. Čítače, jež jsou k dispozici, jsou uvedeny v dolní části souboru *PMCTestA.cpp*. Na Obrázku 2 – Výpis z měřícího programu, jsou vidět čtyři sloupce, z nichž každý znázorňuje měřenou „veličinu.“ Počet řádků je roven počtu opakování měření, které je možné nastavit v souboru *PMCTestB* stejně tak, jako počet vláken či samotné měřené veličiny.



Obrázek 2 - Výpis z měřícího programu

3.4 Time-Stamp Counter

Procesory firmy Intel umožňují programátorům přistoupit k Time-Stamp Counteru. Tento time-stamp counter uchovává přesný počet hodinových cyklu, ke kterým v procesoru dojde. Time-stamp counter je označován jako 64-bitový MSR (model specific register), který je inkrementován každým hodinovým cyklem. Při resetování tohoto čítače, je jeho hodnota nastavena na nulu. Pro přístup k tomuto čítači mohou programátoři využití RDTSC (read time-stamp counter)

instrukci. Tato instrukce načítá vyšších 32 bitů do registru EDX a nižších 32 bitů do registru EAX. Tento čítač by neměl být použit pro měření celého programu, ale pouze pro jeho malé úseky. Tyto úseky pak mohou být mezi sebou porovnávány a měřeny. [11] Následující výpis kódu znázorňuje, jak by mohla vypadat samotná implementace pro měření hodinových cyklů s pomocí time-stamp čítače.

```
__int64 GetCPUCount( unsigned int loword, unsigned int hiword )
{
            _asm
            {
                  _emit 0x0f
                  _emit 0x31
                  mov hiword , edx
                  mov loword , eax
            }
      return ( (__int64) hiword << 32 ) + loword;
}

void main()
{
      unsigned int hi = 0, lo = 0;
      long long int t = GetCPUCount ( lo, hi );

      MyFunction(); //funkce, kterou chceme měřit

      long long int CycleCount = GetCPUCount ( lo, hi ) - t;
      cout << endl;
      cout << CycleCount;
}
```

Výpis kódu 10 - Ukázka měření s pomocí Time-Stamp counteru

_emit je pseudo-instrukce, která definuje jeden bajt v aktuální pozici aktuálního segmentu. Pro definování většího množství bajtů je nutné volat tuto pseudo-instrukci opakovaně. Například takto:

```
__asm _emit 0x0F __asm _emit 0x31 __asm _emit 0xC6
```

Výpis kódu 11 - Ukázka definování _emit instrukce

3.5 Rozdíly v měření hodinového cyklu

Tato podkapitola je věnována rozdílům v měření hodinového cyklu. Bude zde vyjádřen poměr mezi naměřenými hodnotami hodinového cyklu programem od pana Agnera Foga a využitím Time-Stamp čítače. Pokud bychom vzali jednoduchý algoritmus výpisu čísel do konzoly, výsledky by byly pro jednotlivé programy následující. Při pohledu na níže uvedenou tabulku 4 je patrné, že naměřené hodnoty jsou zhruba s desetiprocentní odchylkou.

Tabulka 4 - Porovnání měření hodinových cyklů

Výpis čísel do konzoly	program Agnera Foga	Time-Stamp Counter	Rozdíl mezi druhým a třetím sloupcem
100	24 536 113	27 737 146	88,46%
200	68 696 270	66 719 665	102,96%
300	102 808 740	116 048 332	88,59%
400	143 232 469	145 976 927	98,12%
500	192 565 436	197 469 701	97,52%
1000	430 361 821	406 946 316	105,75%

Rozhodl jsem se otestovat použití Time-stamp čítače i na notebooku pro porovnání naměřené frekvence procesoru (rozdíl mezi notebookem připojeným ke zdroji a s odpojeným zdrojem). Informace o frekvenci procesoru jsem získal z programu CPUID, jenž je volně dostupný ke stažení z internetu. V následující tabulce 5 jsou uvedeny výsledky naměřených hodinových cyklů pomocí Time-stamp counteru. Při pohledu na poslední sloupec tabulky je zřejmé, že počet hodinových cyklů bez napájení notebooku je 3 – 4krát větší, než při měření na stejném notebooku se zdrojem.

Tabulka 5 - Porovnání výkonu CPU notebooku se zdrojem a bez něj

Výpis čísel do konzoly	S napájením	Bez napájení	Procentuální rozdíl
100	63 853 335	264 826 863	414,74%
200	208 329 303	789 456 234	378,95%
300	305 445 724	1 254 734 376	410,79%
400	564 378 766	1 776 334 812	314,74%
500	704 220 563	2 386 866 258	338,94%
1000	1 097 965 638	3 855 656 931	351,16%

Notebook má následující parametry:

```
Počet jader          2
Počet vláken         2
Název                Intel Mobile Core 2 Duo T9600
Specifikace          Intel(R) Core(TM)2 Duo CPU T9600 @ 2.80GHz
Typ paměti           DDR2
Velikost paměti      4 GB
Frekvence paměti     399.0 MHz
```

Program CPUID změřil frekvenci procesoru notebooku jak se zdrojem napájení tak bez něj. V režimu bez napájení byla hodnota procesoru 798 MHz. Po připojení do sítě napětí se hodnota frekvence při výpisu čísel zvýšila na 2 793,5 MHz, což je přibližně třiapůlkrát vyšší, než bez napájení. Přibližně i této hodnotě odpovídá rozdíl mezi hodnotami ve druhém a třetím sloupci tabulky číslo 5. Frekvenci procesorů lze taky změřit v C/C++ programově. Při vypnutém zdroji napájení jsem se dostal na hodnotu 793 MHz a se zdrojem napájení frekvence obou prostorů vzrostla na číslo 2 798 MHz. Lze si povšimnout, že výše uvedení hodnoty se liší v jednotkách Hz tedy jen minimálně.

4. Příklady

Tato kapitola obsahuje rozbor několika příkladů (například algoritmus dvojrozměrné konvoluce). Tento příklad bude rozebrán do detailů, kterými budou posloupnost instrukcí, zobrazení ASM výstupu či pohled na kód pomocí Online Disassembleru a poukázání na rozdíly mezi těmito dvěma postupy. Dále se pak tato kapitola bude zabývat optimalizacemi tohoto algoritmu, a to například pomocí OpenMP direktiv či TBB knihovny. Bude zde rozebrána možnost zvýšení výkonu pomocí návrhového vzoru nesoucí název ObjectPool. Součástí je i ukázka práce s NAN číslem a rozdíl v časech při práci s NAN číslem a bez něj na obecném příkladu i u dvourozměrné konvoluce. Je zde znázorněno i použití SSE instrukcí na algoritmu skalárního součinu.

4.1 Konvoluce

Konvoluce je obecné označení matematického operátoru, jenž zpracovává dvě funkce (například sečtení dvou konečných řad). Ve zpracovávání obrazu se konvolucí myslí použití nějakého filtru na námi zkoumaný obrázek. Tento obrázek je reprezentován maticí s hodnotami. Druhá matice je tzv. konvoluční maska či jádro. Většinou je velikost konvoluční matice 3×3 nebo 5×5. Mezi obrázkem a konvoluční maskou dochází k násobení mezi jednotlivými hodnotami. [15]

4.2 Základní měření konvoluce

V následující uvedené tabulce Tabulka 6 jsou zobrazeny výsledky jednotlivých měření. První sloupec znázorňuje algoritmus konvoluce. První řádek v tomto sloupci popisuje algoritmus s použitím generování a načtení jak konvoluční masky, tak hodnoty (skaláry jasu) obrazu, následné úpravy, algoritmus výpočtu 2D konvoluce a výsledek nových hodnot obrazu vypsaný do konzole. Druhý řádek popisuje tentýž algoritmus ovšem bez generování náhodných hodnot jak masky, tak obrazu. Předposlední řádek je věnován pouze algoritmu konvoluce s výpisem výsledků, bez jednotlivých inicializací, generování a načítání matic. V posledním řádku jsou uloženy hodnoty „čistého“ algoritmu 2D konvoluce bez výpisu do konzole. Obraz je v tomto konkrétním případě reprezentován maticí o velikosti 3×3 a rovněž i maska má velikost matice 3×3.

Druhý sloupec je určený k evidenci doby trvání jednotlivých algoritmů. Třetí sloupec s názvem Clock cycles obsahuje naměřené hodnoty hodinových cyklů. Ve čtvrtém sloupci jsou pak počty jednotlivých provedených instrukcí. Předposlední sloupec, pojmenovaný µOps, čítá vykonané mikrooperace a v posledním sloupci Cache miss jsou uchovány hodnoty Data cache miss. Hodnoty druhého sloupce jsou pořízeny prostřednictvím použití funkce *QueryPerformanceCounter* a zbylé čtyři sloupce obsahují hodnoty získané pomocí programu od pana Agnera Foga.

Tabulka 6 - Výsledky měření konvoluce

Popis algoritmu	Čas	Clock cycles	Instructs	µOps	Cache miss
Celý algoritmus	7,11 ms	11 278 150	1 310 045	2 057 589	34 282
Bez generování souborů	2,86 ms	7 356 610	321 163	529 635	7 330
Konvoluce s výpisem	1,62 ms	4 385 979	173 572	302 009	5 942
Algoritmus konvoluce	0,44 µs	1 036	1752	2 068	0

V následující tabulce Tabulka 7 je vidět procentuální rozdíl mezi jednotlivými měřeními. Druhý sloupec znázorňuje poměr mezi naměřenými hodinovými cykly a taktovací frekvencí procesoru. V posledním sloupci jsou uvedeny procentuální rozdíly mezi hodnotami naměřenými s využitím *QueryPerformanceCounter* a poměrem hodinového cyklu k taktovací frekvenci.

Z Tabulky je patrné, že hodnoty získané měřením jednotlivých částí kódu s využitím funkce *QueryPerformanceCounter* jsou vyšší, než poměr získaný mezi hodinovými cykly a taktovací frekvencí. Tento rozdíl je způsobený pravděpodobně odchylkami při měření funkcí *QueryPerformanceCounter*.

Tabulka 7 – Rozdíl mezi QuerryPerformanceCounter a Clock cycle

Popis algoritmu	Clock cycles/taktovací frekvence	Rozdíl v %
Celý algoritmus	0,006 629 7	6,62
Bez generování souborů	0,002 404 1	15,94
Konvoluce s výpisem	0,001 433 3	11,52
Algoritmus konvoluce	0,000 000 3	23,05

V následující tabulce Tabulka 8 jsou uvedeny jednotlivé výsledky výpočtu konvoluce bez optimalizačních prostředků. Velikost masky je ve všech případech stejná 3×3 a velikost obrazu se různí. Další jednotlivé sloupce pak znázorňují měřené veličiny. Pod sloupcem *čas* jsou hodnoty naměřené pomocí funkce *QueryPerformanceCounter*. Zbylé údaje, jakými je počet hodinových cyklů, instrukcí, mikrooperací a rovněž i cache miss, jsou pak získány pomocí programu od pana Agnera Foga.

Tabulka 8 - Výsledky bez optimalizací

Bez použití optimalizačních prostředků (1 vlákno)						
	Obraz	**čas**	**Clock cycles**	**Instructs**	**µOps**	**Cache miss**
MASKA 3×3	3×3	1,6 µs	4 880	2 298	5 108	0
	5×5	3,4 µs	10 454	4 884	10 954	0
	10×10	13,3 µs	40 723	19 106	42 466	0
	20×20	44,8 µs	139 011	64 350	144 303	16
	50×50	0,29 ms	886 099	416 720	926 345	87
	100×100	1,14 ms	3 491 562	1 637 882	3 641 264	1 730
	1000×1000	0,121 s	370 265 947	173 835 986	386 344 619	181 213

Následující tabulka 9 zobrazuje výsledky měření algoritmu konvoluce s použitím OpenMP direktiv pro paralelizaci kódu. Stejně jako v předchozím případě jsou výsledky měření získány pomocí programu od pana Agnera Foga a funkce *QueryPerformanceCounter*. Lze si povšimnout, že hodnoty v tabulce 8 jsou větší, než v tabulce 9 bez optimalizace. Tento nárůst je způsoben rozdělováním cyklů mezi jednotlivá jádra. Rovněž se tyto rozdělené části programu se na konci slučují do hlavního vlákna a to ukončí program.

Tabulka 9 - použití OpenMP optimalizace

S OpenMP (4 vlákna; 2 fyzická + 2 logická jádra)						
	Obraz	**čas**	**Clock cycles**	**Instructs**	**µOps**	**Cache miss**
MASKA 3×3	3×3	1,2 µs	3 672	1 723	3 844	0
	5×5	3,1 µs	9 846	4 456	9 961	0
	10×10	7,2 µs	22 320	10 343	23 102	0
	20×20	22,8 µs	69 867	32 756	73 123	0
	50×50	0,16 ms	489 566	231 128	512 855	216
	100×100	0,58 ms	1 774 982	833 467	1 849 941	1 721
	1000×1000	0,056 s	171 369 213	80 451 321	179 406 019	199 648

V následující tabulce (tabulka 10) jsou jako v předchozích případech naměřené hodnoty získané prostřednictvím funkce *QueryPerformanceCounter* a programem pana Agnera Foga. Jedná se o měření algoritmu konvoluce za pomocí optimalizace pomocí TBB knihovny.

Tabulka 10 - TBB optimalizace

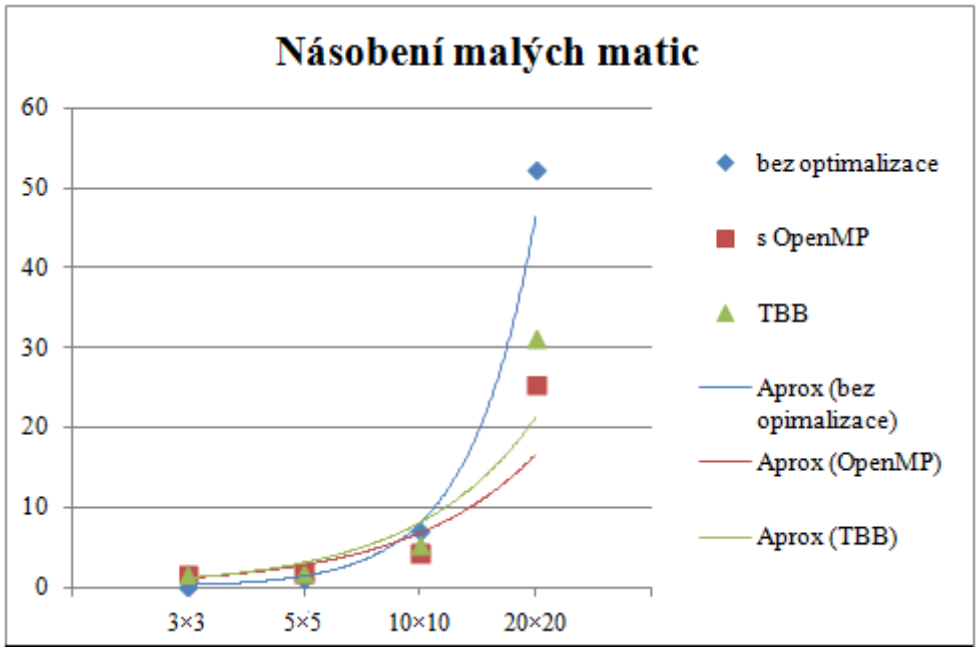
TBB optimalizace (4 vlákna; 2 fyzická + 2 logická jádra)						
	Obraz	**čas**	**Clock cycles**	**Instructs**	**µOps**	**Cache miss**
MASKA 3x3	3×3	1,3 µs	3 978	1 867	4 164	0
	5×5	3,23 µs	9 884	4 644	10 347	0
	10×10	6,9 µs	21 199	9 919	22 106	0
	20×20	21,7 µs	66 444	31 274	70 156	64
	50×50	0,14 ms	42 843	201 239	449 619	223
	100×100	0,56 ms	1 714 594	805 470	1 794 831	1 636
	1000×1000	0,055 s	168 301 220	79 068 284	176 255 480	201 924

V tabulce 11 jsou znázorněny časové výsledky násobení dvou matic pomocí funkce *QueryPerformanceCounter*. Jednotlivé sloupce zobrazují formu optimalizace. Optimalizace pomocí OpenMP direktiv je na malých velikostech matic například 3×3 a 5×5 neúčinná a čas je v těchto případech podstatně větší. Při větších intervalech jako je násobení matic o velikostech 50×50 a 100×100 se projevuje efekt paralelizmu. Čas zde dosahuje zhruba jen padesáti procent.

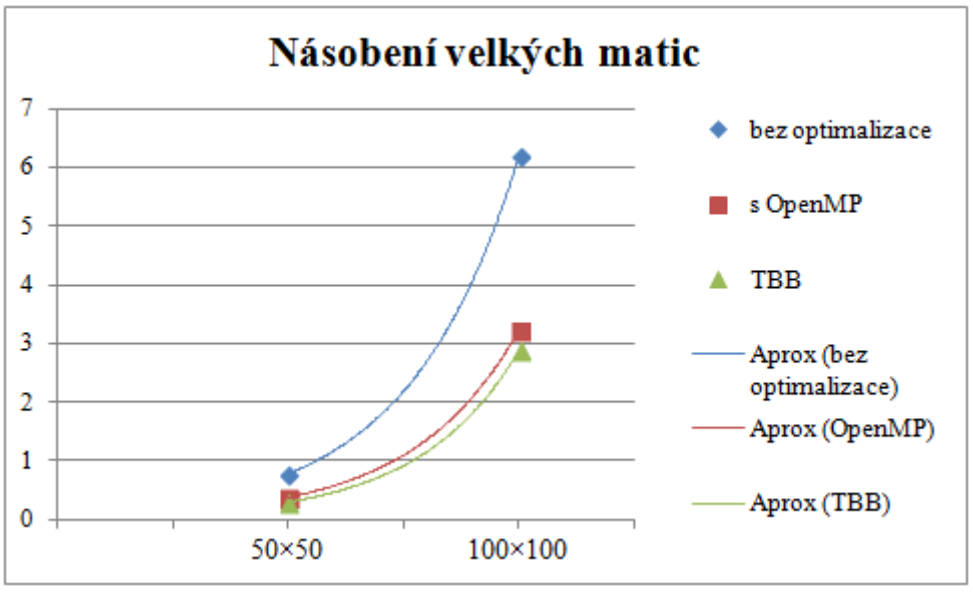
Tabulka 11 - Porovnání násobení matic

Porovnání časových výsledků			
Velikost matice	**bez optimalizace**	**s OpenMP**	**TBB**
3×3	0,27 µs	1,68 µs	1,71 µs
5×5	1,16 µs	1,86 µs	2,05 µs
10×10	7,21 µs	4,34 µs	5,49 µs
20×20	52,28 µs	25,37 µs	31,22 µs
50×50	0,77 ms	0,37 ms	0,29 ms
100×100	6,19 ms	3,22 ms	2,87 ms

Pokud bychom z výše uvedené tabulky vytvořili grafy, vyšla by nám tato následující zobrazení. Na svislých osách jsou zobrazeny časové nároky (na obrázku 3 v mikrosekundách a na obrázku 4 v milisekundách) a na vodorovných osách jsou zase velikosti matic. Oba grafy znázorňují naměřené hodnoty a křivky, jež tyto hodnoty aproximují.



Obrázek 3 - Porovnání násobení matic kratší úseky



Obrázek 4 - Porovnání násobení matic delší úseky

4.3 Disassembly Window

Následující výstup kódu je pořízen v Disassembly módu Microsoft Visual Studia. Odpovídá algoritmu výpočtu dvourozměrné konvoluce.

```
for(i=0; i < index1; ++i)    // rows
mov         dword ptr [ebp-0FCh],0
jmp         main+1DCh (0B168Ch)
mov         eax,dword ptr [ebp-0FCh]
add         eax,1
mov         dword ptr [ebp-0FCh],eax
cmp         dword ptr [ebp-0FCh],3
jge         main+34Ah (0B17FAh)
{
   for(j=0; j < index2; ++j// columns
mov         dword ptr [ebp-108h],0
jmp         main+204h (0B16B4h)
mov         eax,dword ptr [ebp-108h]
add         eax,1
mov         dword ptr [ebp-108h],eax
cmp         dword ptr [ebp-108h],3
jge         main+345h (0B17F5h)
    {
        for(m=0; m < kindex1; ++m)
// kernel rows
mov         dword ptr [ebp-114h],0
jmp         main+22Ch (0B16DCh)
mov         eax,dword ptr [ebp-114h]
add         eax,1
mov         dword ptr [ebp-114h],eax
cmp         dword ptr [ebp-114h],3
jge         main+340h (0B17F0h)
        {
            mm = kindex1 - 1 - m;
mov         eax,2
sub         eax,dword ptr [ebp-114h]
mov         dword ptr [ebp-120h],eax

            for(n=0; n < kindex2;
++n) // kernel columns
mov         dword ptr [ebp-150h],0
jmp         main+265h (0B1715h)
mov         eax,dword ptr [ebp-150h]
add         eax,1
mov         dword ptr [ebp-150h],eax
cmp         dword ptr [ebp-150h],3
jge         main+33Bh (0B17EBh)
            {
                nn = kindex2 - 1 - n;
mov         eax,2
sub         eax,dword ptr [ebp-150h]
mov         dword ptr [ebp-12Ch],eax

             ii = i + (m - kCenterY);
mov         eax,dword ptr [ebp-114h]
sub         eax,dword ptr [ebp-168h]
add         eax,dword ptr [ebp-0FCh]
mov         dword ptr [ebp-144h],eax
            jj = j + (n -kCenterX);
mov         eax,dword ptr [ebp-150h]
sub         eax,dword ptr [ebp-15Ch]
add         eax,dword ptr [ebp-108h]
mov         dword ptr [ebp-138h],eax

                if( ii >= 0 && ii <
index1 && jj >= 0 && jj < index2 )
cmp         dword ptr [ebp-144h],0
jl          main+336h (0B17E6h)
cmp         dword ptr [ebp-144h],3
jge         main+336h (0B17E6h)
cmp         dword ptr [ebp-138h],0
jl          main+336h (0B17E6h)
cmp         dword ptr [ebp-138h],3
jge         main+336h (0B17E6h)

      out[i][j] += in[ii][jj] *
kernelmat[mm][nn];
mov         eax,dword ptr [ebp-0FCh]
imul        eax,eax,0Ch
lea         ecx,[ebp+eax-44h]
mov         edx,dword ptr [ebp-144h]
imul        edx,edx,0Ch
lea         eax,[ebp+edx-0D8h]
mov         edx,dword ptr [ebp-120h]
imul        edx,edx,0Ch
lea         edx,[ebp+edx-0A0h]
mov         esi,dword ptr [ebp-138h]
mov         edi,dword ptr [ebp-12Ch]
mov         eax,dword ptr [eax+esi*4]
imul        eax,dword ptr [edx+edi*4]
mov         edx,dword ptr [ebp-108h]
add         eax,dword ptr [ecx+edx*4]
mov         ecx,dword ptr [ebp-0FCh]
imul        ecx,ecx,0Ch
lea         edx,[ebp+ecx-44h]
mov         ecx,dword ptr [ebp-108h]
mov         dword ptr [edx+ecx*4],eax
            }
jmp         main+256h (0B1706h)
        }
jmp         main+21Dh (0B16CDh)
    }
jmp         main+1F5h (0B16A5h)
}
jmp         main+1CDh (0B167Dh)
```

Výpis kódu 12 - Disassembly window 2D konvoluce

4.4 ASM výstup

Následující výstup kódu roven ASM výstupu. Odpovídá algoritmu výpočtu konvoluce.

```
; 47   : for(i=0; i < index1; ++i)
// rows
       mov     DWORD PTR _i$[ebp], 0
       jmp     SHORT $LN19@main
$LN18@main:
       mov     eax, DWORD PTR _i$[ebp]
       add     eax, 1
       mov     DWORD PTR _i$[ebp], eax
$LN19@main:
       cmp     DWORD PTR _i$[ebp], 3
       jge     $LN17@main
; 48   : {
; 49   :     for(j=0; j < index2; ++j)
// columns
       mov     DWORD PTR _j$[ebp], 0
       jmp     SHORT $LN16@main
$LN15@main:
       mov     ecx, DWORD PTR _j$[ebp]
       add     ecx, 1
       mov     DWORD PTR _j$[ebp], ecx
$LN16@main:
       cmp     DWORD PTR _j$[ebp], 3
       jge     $LN14@main
; 50   :     {
; 51   :         for(m=0; m < kindex1;
++m)     // kernel rows
       mov     DWORD PTR _m$[ebp], 0
       jmp     SHORT $LN13@main
$LN12@main:
       mov     edx, DWORD PTR _m$[ebp]
       add     edx, 1
       mov     DWORD PTR _m$[ebp], edx
$LN13@main:
       cmp     DWORD PTR _m$[ebp], 3
       jge     $LN11@main
; 52   :         {
; 53   :             mm = kindex1 - 1 - m;
// row index of flipped kernel
       mov     eax, 2
       sub     eax, DWORD PTR _m$[ebp]
       mov     DWORD PTR _mm$[ebp], eax
; 55   :             for(n=0; n < kindex2;
++n) // kernel columns
       mov     DWORD PTR _n$[ebp], 0
       jmp     SHORT $LN10@main
$LN9@main:
       mov     ecx, DWORD PTR _n$[ebp]
       add     ecx, 1
       mov     DWORD PTR _n$[ebp], ecx
$LN10@main:
       cmp     DWORD PTR _n$[ebp], 3
       jge     $LN8@main
; 56   :             {
; 57   :                 nn = kindex2 - 1
- n;  // column index of flipped kernel
       mov     edx, 2
       sub     edx, DWORD PTR _n$[ebp]
       mov     DWORD PTR _nn$[ebp], edx
; 59   :                 // index of input
signal, used for checking boundary
; 60   :                 ii = i + (m -
kCenterY);
       mov     eax, DWORD PTR _m$[ebp]
       sub     eax, DWORD PTR
_kCenterY$[ebp]
       add     eax, DWORD PTR _i$[ebp]
       mov     DWORD PTR _ii$[ebp], eax
; 61   :                 jj = j + (n -
kCenterX);
       mov     ecx, DWORD PTR _n$[ebp]
       sub     ecx, DWORD PTR
_kCenterX$[ebp]
       add     ecx, DWORD PTR _j$[ebp]
       mov     DWORD PTR _jj$[ebp], ecx
;; 63   :                 // ignore input
samples which are out of bound
; 64   :                 if( ii >= 0 && ii
< index1 && jj >= 0 && jj < index2 )
       cmp     DWORD PTR _ii$[ebp], 0
       jl      SHORT $LN7@main
       cmp     DWORD PTR _ii$[ebp], 3
       jge     SHORT $LN7@main
       cmp     DWORD PTR _jj$[ebp], 0
       jl      SHORT $LN7@main
       cmp     DWORD PTR _jj$[ebp], 3
       jge     SHORT $LN7@main
       out[i][j] += in[ii][jj] *
kernelmat[mm][nn];
       mov     edx, DWORD PTR _i$[ebp]
       imul    edx, 12
               ; 0000000cH
       lea     eax, DWORD PTR
_out$[ebp+edx]
       mov     ecx, DWORD PTR _ii$[ebp]
       imul    ecx, 12
               ; 0000000cH
       lea     edx, DWORD PTR
_in$[ebp+ecx]
       mov     ecx, DWORD PTR _mm$[ebp]
       imul    ecx, 12
               ; 0000000cH
       lea     ecx, DWORD PTR
_kernelmat$[ebp+ecx]
       mov     esi, DWORD PTR _jj$[ebp]
       mov     edi, DWORD PTR _nn$[ebp]
       mov     edx, DWORD PTR [edx+esi*4]
       imul    edx, DWORD PTR [ecx+edi*4]
       mov     ecx, DWORD PTR _j$[ebp]
       add     edx, DWORD PTR [eax+ecx*4]
       mov     eax, DWORD PTR _i$[ebp]
       imul    eax, 12
               ; 0000000cH
       lea     ecx, DWORD PTR
_out$[ebp+eax]
       mov     eax, DWORD PTR _j$[ebp]
       mov     DWORD PTR [ecx+eax*4], edx
$LN7@main:
; 66   :             }
       jmp     $LN9@main
$LN8@main:
; 67   :         }
       jmp     $LN12@main
$LN11@main:
; 68   :     }
       jmp     $LN15@main
$LN14@main:
```

Výpis kódu 13 - ASM výstup algoritmu konvoluce

4.5 Online Disassembler

Následující výstup je získán z online disassembleru a zobrazuje konvoluční metodu.

```
loc_0041165d:
       mov DWORD PTR [ebp-0x15c],0x1
       mov DWORD PTR [ebp-0x168],0x1
       mov DWORD PTR [ebp-0xfc],0x0
       jmp loc_0041168c

loc_0041167d:
       mov eax,DWORD PTR [ebp-0xfc]
       add eax,0x1
       mov DWORD PTR [ebp-0xfc],eax

loc_0041168c:
       cmp DWORD PTR [ebp-0xfc],0x3
       jge loc_004117fa
       mov DWORD PTR [ebp-0x108],0x0
       jmp loc_004116b4

loc_004116a5:
       mov eax,DWORD PTR [ebp-0x108]
       add eax,0x1
       mov DWORD PTR [ebp-0x108],eax

loc_004116b4:
       cmp DWORD PTR [ebp-0x108],0x3
       jge loc_004117f5
       mov DWORD PTR [ebp-0x114],0x0
       jmp loc_004116dc

loc_004116cd:
       mov eax,DWORD PTR [ebp-0x114]
       add eax,0x1
       mov DWORD PTR [ebp-0x114],eax

loc_004116dc:
       cmp DWORD PTR [ebp-0x114],0x3
       jge loc_004117f0
       mov eax,0x2
       sub eax,DWORD PTR [ebp-0x114]
       mov DWORD PTR [ebp-0x120],eax
       mov DWORD PTR [ebp-0x150],0x0
       jmp loc_00411715

loc_00411706:
       mov eax,DWORD PTR [ebp-0x150]
       add eax,0x1
       mov DWORD PTR [ebp-0x150],eax

loc_00411715:
       cmp DWORD PTR [ebp-0x150],0x3
       jge loc_004117eb
       mov eax,0x2
       sub eax,DWORD PTR [ebp-0x150]
       mov DWORD PTR [ebp-0x12c],eax
       mov eax,DWORD PTR [ebp-0x114]
       sub eax,DWORD PTR [ebp-0x168]
       add eax,DWORD PTR [ebp-0xfc]
       mov DWORD PTR [ebp-0x144],eax
       mov eax,DWORD PTR [ebp-0x150]
       sub eax,DWORD PTR [ebp-0x15c]
       add eax,DWORD PTR [ebp-0x108]
       mov DWORD PTR [ebp-0x138],eax
       cmp DWORD PTR [ebp-0x144],0x0
       jl loc_004117e6
       cmp DWORD PTR [ebp-0x144],0x3
       jge loc_004117e6
       cmp DWORD PTR [ebp-0x138],0x0
       jl loc_004117e6
       cmp DWORD PTR [ebp-0x138],0x3
       jge loc_004117e6
       mov eax,DWORD PTR [ebp-0xfc]
       imul eax,eax,0xc
       lea ecx,[ebp+eax*1-0x44]
       mov edx,DWORD PTR [ebp-0x144]
       imul edx,edx,0xc
       lea eax,[ebp+edx*1-0xd8]
       mov edx,DWORD PTR [ebp-0x120]
       imul edx,edx,0xc
       lea edx,[ebp+edx*1-0xa0]
       mov esi,DWORD PTR [ebp-0x138]
       mov edi,DWORD PTR [ebp-0x12c]
       mov eax,DWORD PTR [eax+esi*4]
       imul eax,DWORD PTR [edx+edi*4]
       mov edx,DWORD PTR [ebp-0x108]
       add eax,DWORD PTR [ecx+edx*4]
       mov ecx,DWORD PTR [ebp-0xfc]
       imul ecx,ecx,0xc
       lea edx,[ebp+ecx*1-0x44]
       mov ecx,DWORD PTR [ebp-0x108]
       mov DWORD PTR [edx+ecx*4],eax

loc_004117e6:
       jmp loc_00411706

loc_004117eb:
       jmp loc_004116cd

loc_004117f0:
       jmp loc_004116a5

loc_004117f5:
       jmp loc_0041167d
```

Výpis kódu 14 - Konvoluce Online Disassembler

4.6 Porovnání výsledků algoritmu konvoluce

V tabulce 3 – Výsledky bez optimalizace jsem si vzal hodnotu ve sloupci Instructs, který značí počet provedených instrukcí. Velikost masky i obrazu jsem zvolil 3x3. V tabulce je napsaná hodnota 3376 instrukcí, jež jsem naměřil pomocí programu od pana Agnera Foga (postup měření je uvedený v podkapitole 3.3). Při manuálním počítání instrukcí z módu Disassembly Window v Microsoft Visual Studiu jsem se dopočítal k hodnotě 3527 instrukcí, což je o 151 instrukcí více. Jedná se tedy přibližně o 4,5% odchylku. Tento relativně malý a při časově nenáročných algoritmech i zanedbatelný rozdíl, může být zcela ignorován. Tato nepřesnost vznikla pravděpodobně měřícím programem.

Pokud se podíváme na Výpis kódu 13 – ASM výstup algoritmu konvoluce a na Výpis kódu 14 – Konvoluce Online Disassembler, je možné si povšimnout téměř totožného výpisu instrukcí. Liší se pouze na začátku Výpisu Online Disassembleru, který obsahuje tři instrukce *mov* následované instrukcí *jmp,* zatímco ASM výstup má pouze jednu instrukci *mov*, rovněž následovanou instrukcí *jmp*. Daly by se očekávat daleko větší rozdíly mezi těmito výpisy.

V následující tabulce 12 jsou uvedeny jednotlivé instrukce, které se vyskytují ve výše uvedených výpisech. Instrukce jsou definovány podle manuálu[1]. Tabulka je rozdělená do čtyř sloupců, kde první značí zkratku instrukce a druhý její stručný popis. Předposlední sloupec Počet µOps znamená, na kolik mikrooperací se jednotlivá instrukce rozpadne. Sloupcem Latence je myšleno zpoždění, které generují instrukce. Cache miss a různé výjimky mohou výrazně zvýšit počet hodinových cyklů. Některá NAN čísla, kterými může být například nekonečno, odmocnina ze záporného čísla, popřípadě číslo, jež není reprezentováno konečnou hodnotou, mají za následek velké zpoždění.

Tabulka 12 - Popis instrukcí

Instrukce	Popis	Počet µOps	Latence
LEA	Provádí výpočet efektivní adresy operandu, načte adresu do registru.	1	1
SUB	Odčítání	1	1
MOV	Kopíruje data z jednoho místa na druhé	1	1
JMP	Skok	1	0
ADD	Sčítání	1	1
CMP	Porovnává operandy	1	1
JGE	Podmíněný skok	1	0
IMUL	Násobení	3	2
JL	Skok v případě, pokud je hodnota menší	1	0
PUSH	Vloží data do zásobníku	2	-
POP	Vyjme data ze zásobníku	1	2

Fakt, že operace s NAN čísly mají za následek velké zpoždění, jsem se rozhodl ověřit. Vzal jsem naprosto jednoduchý algoritmus. Definoval jsem si tři proměnné datového typu float (například *x,y,z*). Do hodnoty *x* jsem uložil libovolnou hodnotu. Hodnotu *y* jsem nastavil na 0 nebo

1 podle toho, zda jsem chtěl pracovat s NAN číslem (hodnota 0) nebo ne (hodnota 1). Poté jsem změřil samotný výpočet hodnoty *z*, jež jsem definoval jako podíl hodnot *x/y*. V obou dvou případech vycházela hodnota téměř totožná. Rozhodl jsem se tedy například provést patnáct základních operací s čísly (sčítání, odčítání, násobení a dělení). Výsledek se dostavil vzápětí. Pro hodnotu *y=0* (při které se počítalo tedy s NAN číslem) byl čas potřebný k výpočtu roven 2,089 mikrosekund. Jakmile jsem změnil hodnotu *y* z nuly na jedničku, dosáhl jsem na časovou hodnotu 0,402 mikrosekund. Tato hodnota odpovídá přibližně pětinovému spotřebovanému času, tedy nějakým 19,24 % oproti práci s NAN číslem. Lze tedy předpokládat, že čím více se pracuje s NAN čísly, tím je rozdíl markantnější i na tak krátkém časovém úseku.

Také jsem vyzkoušel výpočet 2D konvoluce s NAN číslem na velikosti matic 3×3 (konvoluční maska a původního obraz). Samotný výpočet 2D konvoluce bez čísla NAN trval 1,688 mikrosekund. Když jsem jeden prvek v matici změnil na NAN číslo, čas výpočtu konvoluce vzrostl na 4,322 µs, což je 256% rozdíl. Výpočet algoritmu tedy je nyní dvaapůlkrát časově náročnější. Když jsem vyzkoušel změnit tři čísla na NAN, časový výpočet vzrostl na 8,146 µs, jež se rovná 482,5% rozdílu. Je patrné, že čím větší bude matice reprezentující obraz a čím více bude NAN čísel při tomto výpočtu, bude to mít za následek větší časové rozdíly.

4.7 Návrhový vzor ObjectPool

Alokace a dealokace objektů zabírá mnoho času. Pokud program používá mnoho malých dynamicky alokovaných objektů, je dobré zvážit možnost spojit všechny objekty do jednoho kontejneru s výhodou spojité paměti [1]. Existuje návrhový vzor ObjectPool. Ten použijeme právě ve chvílích, kdy potřebujeme omezit z nějakého důvodu počet instancí. Místo vytváření nových instancí se proto dává přednost znovupoužití dříve vytvořených objektů. Klasickým příkladem může být připojení k databázi. ObjectPool tedy obsahuje námi vytvořené instance a ty se v případě nutnosti použijí. Ve chvíli, kdy již není objekt zapotřebí, vrátíme jej zpět do poolu. Pokud se objeví více požadavků, než je počet dostupných instancí, v poolu se již žádný volný objekt nenachází, nepokryté požadavky se poté řadí do fronty a zde čekají, až se některá z instancí uvolní [12].

V následující tabulce jsou uvedeny jednotlivé časové výsledky a jejich procentuální rozdíl při alokaci a dealokaci objektů. První sloupec označuje počet objektů, druhý sloupec znázorňuje uplynulý čas při alokaci a dealokaci objektů bez použití návrhového vzoru ObjectPool. Předposlední sloupec je věnovaný výsledkům při měření času právě s použitím návrhového vzoru ObjectPool a poslední sloupec vyjadřuje procentuální rozdíly mezi druhým a třetím sloupcem. Čas je uváděn ve všech případech v mikrosekundách a použitá funkce, která čas měřila, je *QueryPerformanceCounter*.

Existuje mnoho způsobů, jak ověřit tvrzení, že používání ObjectPoolu je efektivnější, než alokovat a nealokovat paměť. Například stačí pouze alokovat a dealokovat paměť pro obyčejné datové typy, jako může být *char, integer, float* či *double*. Například pokud bychom alokovali a nealokovali sto objektů typu *double* milionkrát po sobě, dostali bychom se na hodnotu s využitím ObjectPoolu na 374 milisekund. Kdybychom alokovali a dealokovali stejný počet objektů bez použití výše zmíněného návrhového vzoru s operátory *new* a *delete*, dostali bychom se na hodnotu uplynulého času 6 958 milisekund.

4.8 Zarovnání dat

Všechna data v RAM paměti by měla být zarovnána na adresy dělitelné mocniny čísla 2 podle následujícího schématu.

Tabulka 13 - Schéma zarovnání dat

Velikost operandu	Zarovnání
1 – byte	1
2 – word	2
4 – dword	4
6 – fword	8
8 – qword	8
10 – tbyte	16
16 – oword, xmmword	16
32 – ymmword	32

Většina starších mikroprocesorů má nevýhodu v tom, že pokud se snaží přistoupit k nezarovnaným datům, stojí je to několik hodinových cyklů. Moderní procesory zvládnou pracovat s nezarovnanými daty téměř stejně rychle jako se zarovnanými. Většina XMM instrukcí, jež čtou a zapisují 16-bajtové operandy požadují, aby byly operandy zarovnány na 16 bajtů. Instrukce, které akceptují nezarovnané 16-bajtové operandy, mohou být poněkud neefektivní na starších procesorech. Tato nevýhoda je kompenzována AVX instrukční sadou. AVX instrukce nevyžadují zarovnání paměťových operandů až na některé výjimky. Procesory, které podporují AVX instrukční sadu obecně zvládnou pracovat s nezarovnanými daty stejně efektivně jako se zarovnanými. [1]

Vezměme v úvahu jednoduchý algoritmus, který bude mít za úkol počítat skalární součin dvou vektorů. Rychlost algoritmu otestuji na FPU koprocesoru a také provedu porovnání rychlosti této operace s použitím SSE instrukcí mezi zarovnanými daty a nezarovnanými. Pro použití SSE instrukcí je zapotřebí mít nastaveno v Microsoft Visual Studiu ve vlastnostech projektu v menu *C/C++*, submenu *Code Genreation* vlastnost *Enable Enhanced Instruction Set* například na hodnotu *Streaming SIMD Extensions 2 (/arch:SSE2) (/arch:SSE2)* (záleží podle podporované verze).

V měření algoritmu skalárního součinu s výše uvedenými variantami jsem dosáhl následujících výsledků. Pro skalární součin bez SSE instrukcí a jakékoliv optimalizace jsem naměřil hodnotu 0,537 mikrosekund. Pro nezarovnaná data s využitím SSE instrukcí jsem se dostal k času 0,431 mikrosekund a s použitím zarovnaných dat se výsledný čas rovnal hodnotě 0,404 mikrosekund. Je zřejmé, že využití SSE instrukcí je velmi výhodné pro tuto operaci.

5. Shrnutí

Tato kapitola obsahuje jednotlivá doporučení, rady a postupy, jak daný kód optimalizovat. Vychází z předchozích kapitol a jejich výsledků měření. Kapitola 3 je věnována analýze měření času. Z výše dosažených výsledků měření je patrné několik následujících tvrzení. Při porovnání měření hodinových cyklů mezi programem od pana Agnera Foga a Time-Stamp čítačem je celkový rozdíl maximálně deset procent. Doporučuji použít oba dva tyto čítače. Samotná implementace pro počítání hodinových cyklů pomocí Time-Stamp counteru je velmi snadná a rychlá i pro začátečníky. Pokud nám jde o komplexnější informace, jakými například může být počet mikrooperací, instrukcí a cahce missy, program od pana Agnera Foga bude nejvhodnější. Je ovšem zapotřebí mít na paměti, že tyto čítače je doporučeno používat v případech, kdy nám nejde o profilování celého programu, ale jen určité části kódu.

Vhodná metoda, která se osvědčila při měření času, je *QueryPerformanceCounter*. Její odchylky při měření krátkých úseků jsou minimální, zatímco u ostatní metod použitých v kapitole 3 tyto odchylky dosahují daleko vyšších hodnot. Musíme si také uvědomit, že při měření hodinových cyklů je dobré tento kód několikrát zopakovat a nikdy neuvažovat první naměřenou hodnotu. Ta je vždy značně vyšší. Je to zapříčiněno tím, že při prvním spuštění kódu nejsou data ani kód v mezipaměti. To má za následek, že počet hodinových cyklů je značně vyšší kvůli cache missům. Teprve až následující měření zobrazují výsledek s uložením dat do mezipaměti. Také doporučuji pro počítání hodinových cyklů a času pomocí funkce *QueryPerformanceCounter* vypnout v `Release` módu veškeré formy optimalizace překladačem.

Důležitou roli při vykonávání algoritmu také hraje vmísení NAN čísel do výpočtu. Považuji za vhodné si dát pozor na toto úskalí a vyvarovat se počítání s těmito čísly. Tyto operace s NAN čísly pak mají za následek značné zpoždění. Pří výpočtu algoritmu konvoluce je při velkých obrazech vhodné použít paralelismus pro urychlení tohoto algoritmu. Nabízí se využít jak OpenMP technologii tak Threading Building Blocks, jejichž výsledky pro optimalizaci obrazů 1000×1000 s konvoluční maskou 3×3 jsou velmi podobné. Provádět optimalizaci pro malé obrazy například 3×3, 5×5 či 10×10 je poněkud zbytečné. V těchto případech dokonce se může stát, že čas vzroste. To je dáno u OpenMP tím, že je zapotřebí rozdělit a sloučit jednotlivá vlákna programu. Dobrým krokem je použít OpenMP při paralelizaci smyček a to především kvůli jednoduchosti a nenáročnosti na znalost této technologie programátorem. Pro komplexnější paralelizaci algoritmů je spíš vhodnější technologie TBB byť je náročnější na znalosti a je zde nutný daleko větší zásah programátora do samotného kódu.

I přes velký výkon dnešních moderních procesorů dochází při vytváření cyklů k nárůstu vykonávaného času. Je tedy zapotřebí si rozmyslet, kdy je dobré cyklus použít a kdy ne. V podkapitole 2.8 v tabulce 1 je znázorněno manuální rozbalování smyček. Nutností je zvážit, jak moc smyčku rozvinout, jelikož tím dochází k nárůstu zdrojového kódu a možnosti chyb i delšího ladění programu. Také se musí zvážit eventualita, že nad těmito rozbalenými částmi smyčky je možné použít SSE instrukce pro provedení společné operace. Nepříjemnost nastane, když je počet opakování smyčky roven například prvočíslu, nebo takovému číslu, které při pozdějších optimalizacích neumožní například využít SSE instrukce.

Při použití SSE instrukcích je dobré mít vždy zarovnané hodnoty na 16 bajtů, práce s nimi je pak daleko efektivnější než s nezarovnanými operandy. Ovšem i práce s nimi je pořád rychlejší

než nechat výpočet (například skalární součin) na FPU. Pro porovnávání optimalizovaného kódu doporučuji si vygenerovat ASM výstup či se přepnout v prvotní fázi do Disassembly Window a podívat se na instrukce, jež se budou vykonávat. Rozdíly v počtu a typu provedených instrukcí, podle formy optimalizace, by na této úrovni už měly být zřejmé. Při malých částech kódu je možné se i manuálně dopočítat, kolik se instrukcí provedlo. Pro zpětnou kontrolu optimalizovaného kódu doporučuji vyzkoušet některý z disassemblerů, například Online Disassembler. V mém případě algoritmu dvourozměrné konvoluce se ASM výstup a výstup z Online Disassembleru od sebe příliš nelišily.

I samotné nastavení kompilátoru je důležitým faktorem při optimalizaci kódu. V *Release* módu si můžeme ve vlastnostech projektu v menu *C/C++* a submenu *Optimization* zvolit formu optimalizace. Máme k dispozici optimalizaci překladačem a sami ji můžeme zakázat nebo povolit a vybrat si vhodnou míru optimalizace podle potřeby. Rovněž jsou zde i možnosti zvolit si podporu inline funkcí, intrinsic funkcí nebo si vybrat mezi tím, zda dáme přednost rychlosti nebo velikosti výsledného kódu.

6. Závěr

Tato poslední kapitola je věnována závěru, jenž má shrnout cíle této bakalářské práce a jednotlivé kapitoly. Cílem této bakalářské práce je vytvořit sadu doporučení a postupů, jež jsou založené na výsledcích měření pro optimalizaci zdrojového kódu.

První kapitola pojednávala především o termínech, jež jsou použity napříč celou prací a různým programovým prostředkům, technologiím pro urychlení vykonávaného kódu. Druhá kapitola byla věnována definicím mařených hodnot a též například využití technologie rozbalování smyček. Ve třetí kapitole analyzuji měření času. Je zde znázorněno několik metod, jak jej změřit, a rozdíly mezi nimi. Také je v této kapitole demonstrován příklad, jak získat měřené veličiny s využitím jednotlivých čítačů pro měření a rozdíly například v měření hodinových cyklů různými čítači. Ve čtvrté kapitole je rozebrán příklad algoritmu dvourozměrné konvoluce a porovnání mezi jednotlivými optimalizačními technologiemi, jež jsou pro tento algoritmus vhodné k paralelizaci. V předposlední kapitole – páté jsou uvedeny rady a doporučení pro optimalizaci kódu, jež vycházejí z výsledků měření.

Použitá literatura:

[1] Fog, Agner. Software optimalization resources. Dostupné z WWW: http://www.agner.org.

[2] Bandyopadhyay, Shibdas; A Study on Performance Monitoring Counters in x86-Architecture. Dostupné z WWW: http://www.cise.ufl.edu/~sb3/files/pmc.pdf

[3] Ličev L.: Architektura počítačů I, skriptum FEI VŠB TUO, 1999. Architektura počítačů I. Dostupné z WWW: http://www.cs.vsb.cz/licev/arp1.pdf

[4] Rechistov, Grigory. Branch prediction. 2011.

[5] Intel Corp. Intel® 64 and IA-32 Architectures Software Developer Manuals. Dostupné z WWW: www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[6] Intel Corp. Intel® 64 and IA-32 Architectures Optimization Reference Manual. Dostupné z WWW: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

[7] Khronos Corp. The open standard for parallel programming of heterogeneous systems. Dostupné z WWW: https://www.khronos.org/opencl/

[8] NVIDIA Corp. Parallel Thread Execution ISA Version 4.1. Dostupné z WWW: http://docs.nvidia.com/cuda/parallel-thread-execution/#axzz3HJw5JEOt

[9] Anon. What Does it Mean to Be CPU Bound? Dostupné z WWW: http://www.wisegeek.com/what-does-it-mean-to-be-cpu-bound.htm

[10] Microsoft Corp. Parameter Passing Dostupné z WWW: http://msdn.microsoft.com/en-us/library/zthk2dkh%28v=vs.100%29.aspx

[11] Anon. Using the RDTSC Instruction for Performance Monitoring. Dostupné z WWW: https://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf

[12] PECINOVSKÝ, Rudolf. Návrhové vzory. Brno: ComputerPress, 2007. ISBN 978-80-251-1582-4.

[13] Intel Corp. Threading Building Blocks. Dostupné z WWW: https://www.threadingbuildingblocks.org/

[14] Gimp Corp. The GIMP, uživatelská příručka. Dostupná z WWW: http://docs.gimp.org/2.2/cs/index.html

[15] Anon. White Paper:Processor Affinity Multiple CPU Scheduling. Dostupné z WWW: http://www.tmurgent.com/WhitePapers/ProcessorAffinity.pdf