FAKULTA INFORMAČNÍCH TECHNOLOGIÍ, VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Architektura procesorů (ACH 2015)
Projekt č. 1: Optimalizace sekvenčního kódu

Radek Hrbáček (ihrbacek@fit.vutbr.cz)

Termín odevzdání: 23. 11. 2015

1 ÚVOD

Cílem tohoto projektu je vyzkoušet si optimalizaci sekvenčního kódu zejména pomocí vektorizace. Projekt je rozdělen na dvě části – v první části bude úkolem dle zadaného postupu optimalizovat úlohu násobení matice a vektoru, v druhé části pak svými vlastními silami vylepšit úlohu částicového systému. Pro analýzu výkonnosti je připravena knihovna PAPI umožňující přístup k zabudovaným hardwarovým *performance counterům* uvnitř procesoru. Veškerý kód bude spouštěn na superpočítači Anselm.

2 SUPERPOČÍTAČ ANSELM

Superpočítač Anselm umístěný na VŠB v Ostravě je složen z celkem 209 uzlů, každý uzel disponuje 2 procesory Intel Xeon, většina (180) uzlů je založena na procesorech Intel Xeon E5-2665. Tyto procesory mají 8 jader s mikroarchitekturou Sandy Bridge, podporují tedy vektorové instrukce AVX. Pro připojení na superpočítač Anselm je potřeba mít vytvořený účet, s kterým je možné se připojit na tzv. čelní (login) uzel – `anselm.it4i.cz`. Tento uzel **neslouží** ke spouštění náročných úloh, veškeré experimenty je nutné provádět na výpočetních uzlech. Pro účely tohoto projektu je nejjednodušším řešením vytvořit *interaktivní úlohu*, např. pomocí následujícího příkazu:

```
[ihrbacek@login1.anselm ~]$ qsub -A IT4I-10-3 -q qexp -l select=1:ncpus=16,walltime=1:00:00 -I
qsub: waiting for job 262806.dm2 to start
qsub: job 262806.dm2 ready

[ihrbacek@cn117 ~]$
```

Příkaz `qsub` zadá požadavek na spuštění úlohy do fronty, jakmile bude v systému dostatek volných uzlů, dojde ke spuštění úlohy. Parametr `-A` určuje projekt, v rámci kterého máme pro tento projekt alokované výpočení hodiny (neměnit), `-q` určuje frontu, do které bude úloha zařazena (pokud nebude úloha dlouhou dobu spuštěna, můžete použít frontu `qprod`, ale preferujte `qexp`), parametr `-l` určuje zdroje, které budou úloze přiděleny (počet uzlů, počet procesorů, čas). Abyste předešli zkreslení výkonových statistik, vždy alokujte celý uzel, tj. 16 jader. Interaktivní úlohu pak získáte parametrem `-I`.
Software na superpočítači Anselm je dostupný pomocí tzv. *modulů*. Tyto moduly je potřeba před použitím načíst. V tomto projektu budou potřeba moduly `intel` a `papi`:

```
module load intel/15.3.187
module load papi
```

Číslo verze modulu za lomítkem je nepovinné, avšak v současnosti je výchozí Intel kompilátor `intel/13.5.192`, proto doporučuji explicitně verzi uvést. Tyto příkazy je nutné spustit při každém spuštění interaktivní úlohy! Modul `intel` zahrnuje `C/C++` kompilátor firmy Intel, který je možné vyvolat příkazy `icc` resp. `icpc`.
Modul `papi` pak obsahuje knihovnu PAPI, která usnadňuje přístup k hardwarovým performance counterům uvnitř procesoru. Každý procesor obsahuje několik (4-8) HW registrů, které jsou schopny počítat předem definované události. Mezi typické události, které nás zajímají, patří počet vykonaných FP/INT/LS instrukcí, IPC, počet přístupů do jednotlivých pamětí cache, propustnost paměti, přesnost predikce skoků atd. Knihovna PAPI obsahuje několik pomocných programů (`papi_avail`, `papi_native_avail`, `papi_mem_info`, ...), pomocí kterých je možné zjistit detaily o podpoře na daném procesoru. Pro zjednodušení práce s knihovnou PAPI je v projektu použita třída PapiCounter, která obaluje knihovnu PAPI. Její definice se nachází v souboru `papi_counter.h`. Kostra obou částí projektu již tuto třídu využívá. Seznam událostí, které chceme měřit, se předává přes proměnnou `PAPI_EVENTS`. Ta je již přednastavena v souboru `Makefile`. Po spuštění programu dojde k vypsání změřených hodnot do konzole.

3 Součin matice a vektoru (5 bodů)

Začněte s naivní implementací v adresáři `matvec/step0` a postupujte dle následujících kroků. Pro každý další krok zkopírujte výsledek posledního kroku do nového adresáře (např. po nultém kroku zkopírujte `matvec/step0` do `matvec/step1`) a pokračujte dalším krokem.

```
void mat_vec_mul(int rows, int cols, float a[][cols], float b[cols], float c[↩
    rows])
{
    int i, j;

    for (i = 0; i < rows; i++)
    {
        c[i] = 0.0f;
        for (j = 0; j < cols; j++)
            c[i] += a[i][j] * b[j];
    }
}
```

3.1 KROK 0: VYPNUTÉ OPTIMALIZACE

Výchozí nastavení kompilátoru používá optimalizace `-O2`. Nás však zajímá, jaký výkon má naivní kód bez využití optimalizací a proto optimalizace vypneme parametrem `-O0`. Prostudujte soubor `Makefile`, kód zkompilujte pomocí příkazu `make` a spusťte pomocí `make run`. Pro detailní pochopení práce kompilátoru prostudujte kód v symbolických instrukcích (`matvec.s`). Najděte kód odpovídající vnitřní smyčce a určete, jaká instrukce byla kompilátorem použita např. pro násobení.

3.2 KROK 1: ZAPNUTÍ OPTIMALIZACÍ (1 BOD)

Nyní zapněte optimalizace (`-O2`) a sdělte kompilátoru, že chcete využít instrukci AVX (`-xavx`), parametry zapište do `Makefile` na řádek `OPT=`. Porovnejte výkonnost s předchozím krokem. Pro zobrazení informací o provedených optimalizací použijte volbu `-opt-report=5`[1]. Parametr zapište do souboru `Makefile` na řádek `REPORT=`, při kompilaci bude vytvořen soubor `matvec.optrpt` obsahující detailní informace o provedených optimalizacích.
Nyní odstraňte direktivu `#pragma nounroll` ve vnitřní smyčce `mat_vec_mul()` a spusťte kompilaci znovu. Najděte v reportu informaci o tom, zda a kolikrát byla smyčka rozbalena. Opět najděte kód odpovídající vnitřní smyčce a určete, jaké instrukce byly kompilátorem použity tentokrát.

3.3 KROK 2: DATOVÉ ZÁVISLOSTI (1 BOD)

V předchozím kroku byly zapnuty optimalizace, došlo však k vektorizaci kódu? To lze zjistit pomocí parametru `-opt-report` (zapište do `Makefile` na řádek `REPORT=`). Z výpisu je patrné, že k vektorizaci nedošlo:

```
remark #15344: loop was not vectorized: vector dependence prevents vectorization
```

Kompilátor neví, jestli cyklus náhodou neobsahuje datové závislosti (pole `a`, `b`, `c` se mohou obecně překrývat). My však víme, že neobsahuje. Můžeme tedy kompilátoru pomoci tím, že mu tuto skutečnost poradíme. To lze provést více způsoby:

- parametr `-fargument-noalias`
- direktiva `#pragma ivdep`
- klíčové slovo `restrict`

Vyberte si libovolnou z variant a znovu zkompilujte kód. Pomocí `-vec-report3` ověřte, že k vektorizaci došlo. Porovnejte výkon s předchozím krokem. Srovnejte vygenerovaný kód v symbolických instrukcích (jaké instrukce byly použity?). Najděte vnitřní smyčku a vysvětlete, proč je kód tak dlouhý.

[1] Více informací o formátu reportu lze nalézt zde: https://software.intel.com/en-us/articles/getting-the-most-out-of-your-intel-compiler-with-the-new-optimization-reports

3.4 Krok 3: zarovnání dat v paměti (1 bod)

Zarovnáním dat v paměti lze dále optimalizovat kód. Pomocí direktiv `__declspec(align(N))` a `__assume_aligned(var, N)` vynuťte zarovnání polí `a`, `b`, `c` v paměti. Na kolik bytů musí být data v paměti zarovnána při použití instrukcí AVX? Opět porovnejte výkon oproti předchozímu kroku. Prohlídněte si také kód v symbolických instrukcích a vysvětlete změny ve vnitřní smyčce. Z hodnot HW counterů určete, zda program využívá vektorovou jednotku efektivně, nebo stále využívá i skalární instrukce.

3.5 Krok 4: padding (1 bod)

Matice `c` má 32 řádků a 31 sloupců, řádky tedy nejsou zarovnané na velikost registrů `YMM`. Změňte parametr `PADDING` v souboru `Makefile` tak, aby byla velikost řádku pole `c` násobkem velikosti registrů `YMM`. Rozměry matice `c` zůstanou zachovány, každý řádek však bude obsahovat určitý okraj, se kterým se nebude počítat. Porovnejte výkon s předchozím krokem a rozdíl vysvětlete. Změnila se nějak vnitřní smyčka? Vysvětlete.

3.6 Krok 5: zarovnání vektorů (1 bod)

Nyní již máme velmi optimalizovaný kód, kompilátor však stále předpokládá, že délka řádku pole `c` nemusí být násobkem velikosti registrů `YMM`. Poraďte kompilátoru, že jednotlivé řádky jsou zarovnány - využijte direktivu `#pragma vector aligned`. Opět porovnejte výkon a kód v symbolických instrukcích.

3.7 Výstup první části projektu

Výstupem první části projektu budou zdrojové kódy jednotlivých kroků, tedy adresáře `step0`, `step1`, ..., `step5`, a soubor `matvec.txt` s textovým komentářem k jednotlivým krokům – u každého kroku vyplňte naměřené parametry výkonu a odpovězte na všechny otázky.

4 Částicový systém (10 bodů)

Cílem této části projektu bude nejprve implementovat a posléze optimalizovat výpočet vzájemného silového působení N těles. Každé těleso má jistou hmotnost, polohu v prostoru a rychlost. Gravitační síly působící na dané těleso od ostatních těles mají různé směry a jejich výslednice způsobuje změnu rychlosti tohoto tělesa. Pro vektory polohy, rychlosti a zrychlení platí:

$$\mathbf{r}^{i+1} = \mathbf{r}^{i} + \mathbf{v}^{i+1} \cdot \Delta t \tag{4.1}$$

$$\mathbf{v}^{i+1} = \mathbf{v}^{i} + \mathbf{a}^{i+1} \cdot \Delta t \tag{4.2}$$

$$\mathbf{a}^{i+1} = \frac{\sum \mathbf{F}_j^{i+1}}{m} \tag{4.3}$$

Síla $\mathbf{F}^{i+1}$ působící na těleso je dána vektorovým součtem dílčích sil způsobených gravitačním polem ostatních těles. Dvě tělesa na sebe působí gravitační silou danou:

$$F = \frac{G \cdot m_1 \cdot m_2}{R^2}, \tag{4.4}$$

kde $G = 6.67384 \cdot 10^{-11}\mathrm{Nm}^2\mathrm{kg}^{-2}$ je gravitační konstanta, m_1 a m_2 jsou hmotnosti těles a R je jejich vzdálenost. V každém kroku výpočtu je tedy nutné spočítat síly působící mezi jednotlivými tělesy a změny rychlostí a poloh jednotlivých těles.

4.1 Krok 0: základní implementace (3 body)

Kostra aplikace je připravena v adresáři `nbody/step0`. Prvním krokem bude doplnění funkce `particles_simulate` v souboru `nbody.cpp` tak, aby funkce správně simulovala pohyb `N` částic `particles` v `STEPS` krocích s časovým posuvem `DT` sekund. Všechny tyto parametry jsou definovány v souboru `Makefile`. Správnost výpočtu je možné ověřit porovnáním výstupního souboru se vzorovým výstupem `nbody/output.dat`.

4.2 Krok 1: optimalizovaná implementace (4 body)

Jakmile bude implementace funkční, zkopírujte celý adresář `nbody/step0` do nového adresáře `nbody/step1` a pokračujte optimalizací kódu. Porovnejte výkon základní implementace s optimalizovaným kódem. Do textového souboru `nbody.txt` popište provedené optimalizace a vysvětlete jejich dopad. Pokud nelze kód dále optimalizovat pomocí předvedených metod, zdůvodněte.

4.3 Krok 2: analýza paměťové hierarchie (3 body)

Zkopírujte celý adresář `nbody/step1` do nového adresáře `nbody/step2`. Modifikujte proměnnou `PAPI_EVENTS` v `Makefile` tak, abyste mohli zjistit míru výpadků v jednotlivých úrovních cache (L1, L2, L3). Statistiky jednotlivých úrovní cache sledujte postupně, počet použitelných HW counterů je omezen. Použité HW countery zapište do `nbox.txt`. Pomocí programu `gen` generujte datové soubory různých velikostí. Např. pro vygenerování souboru s 20000 částicemi použijte následující příkaz:

```
./gen 20000 20k.dat
```

Sledujte míru výpadků jednotlivých úrovní cache a určete tak hranice, kdy velikost dat přesáhne velikost cache L1 a L2. Teoreticky určete hranici pro cache L3. Výsledky zapište do souboru `nbody.txt`. Popište, jaký dopad na výkon mají výpadky v cache a jak byste tento vliv omezili.

5 Výstup projektu a bodování

Výstupem projektu bude soubor `xlogin00.zip` obsahující všechny zdrojové soubory a textové soubory `matvec.txt` a `nbody.txt` obsahující textový komentář k oběma částem pro-

jektu. V každém souboru nezapomeňte uvést svůj login! Hodnotit se bude jak funkčnost a správnost implementace, tak textový komentář – ten by měl dostatečně popisovat rozdíly mezi jednotlivými kroky a odpovídat na otázky uvedené v zadání. Projekt odevzdejte v uvedeném termínu do informačního systému.