TWS: Optimalisatie

Gowri Suryanarayana

8 december 2015

1 Oefeningen voor tijdens de oefenzitting

1.1 De floating point stack en het effect van optimalisaties op intel

Als je weet dat een `float` ongeveer 7 decimale cijfers heeft, wat verwacht je dan voor de waarde van `w` in het volgende fragment?

```
float x = 1;
float y = pow(2.0f, -25.0f);
float z = 1;
float w = (x + y) - z;
```

M.a.w. we willen $w = (1 + 2^{-25}) - 1 = 2^{-25} \approx$ `2.98023224e-08` uitrekenen. Experimenteer hiervoor eens met het programma `fpu.cpp`.

1. Compileer zonder optimalisatie (`-O0`) en met `-mfpmath=387` . Ben je verrast? Bekijk de assembler output m.b.v.

   ```
   objdump -D fpu | c++filt | less
   ```

   Zoek nu naar de string `f<float` m.b.v. `/`, met `n` (next) ga je naar de volgende match (met `N` naar de vorige). (Met `q` quit je `less`.) Zorg dat je hiermee de assembler code van de functie `f` voor `float` terug vindt (m.a.w. de functie `float<f>`). Je vindt iets terug als

   ```
   8048b17:       d9 45 fc                flds   -0x4(%ebp)       # float load
   8048b1a:       d8 45 f8                fadds  -0x8(%ebp)       # float add
   8048b1d:       d8 65 f4                fsubs  -0xc(%ebp)       # float substract
   ```

   Alle bewerkingen gebeurden dus op de floating point stack. De floating point stack op een intel machine werkt steeds in "extended precision" met 80 bit floats, dit zijn in feite `long double`'s.

   Merk op dat dit niet noodzakelijk goed is. Het kan bijvoorbeeld voor problemen zorgen wanneer je Kahan summation implementeert (of de `sumK` van de eerste oefenzitting). (In zo'n gevallen kan soms het keyword `volatile` je uit de nood helpen.)

2. Compileer nu opnieuw zonder optimalisatie maar voeg ditmaal `-ffloat-store` toe. Wat is nu het resultaat? Ga opnieuw m.b.v. `objdump` op zoek naar de gegeneerde assembler code. Je zou nu `fstps` instructies moeten vinden. Wat doen die? (Maak eventueel gebruik van Prof. Google.)

3. Compileer m.b.v. `-O1` en ga opnieuw op zoek naar de functie. Wat merk je? Kijk ook naar de assembler instructies voor `f<double>` en `f<long double>`...

4. Compileer m.b.v. `-O3` en doe hetzelfde. Wat gebeurt er?

5. Compileer nu met `-O0 -msse4 -mfpmath=sse`. Wat gebeurt er nu? Wat voorspel je voor de hogere optimalisaties? Test of je het juist hebt. (Een ander alternatief zou ook nog `-mpc32` zijn, maar dan gebeuren alle operaties op de FPU met 32 bits, ook die voor `double` en `long double` argumenten.)

1.2 Het gebruik van de TSC op x86

Voor het nauwkeurig timen kan je op heel wat architecturen gebruik maken van *hardware performance counters*. Op de intel architectuur kan dit door middel van de *time stamp counter* (TSC). De assembler instructie om deze uit te lezen is simpelweg `rdtsc` en plaatst een 64 bit unsigned integer in de registers `EDX:EAX`. De GNU compiler en de intel compiler kunnen de TSC uitlezen met behulp van volgende inline assembly:

```
inline uint64_t rdtsc()
{
    uint32_t lo, hi;
    asm volatile ("rdtsc\n\t" : "=a" (lo), "=d" (hi));
    return (uint64_t) hi << 32 | lo;
}
```

De waarde die je terug krijgt is het aantal clock cycles. Als je de snelheid van je processor weet (b.v. met `cat /proc/cpuinfo | grep 'cpu MHz'`), dan kan je de tijd voor 1 clock cycle vinden als S^{-1} met S de snelheid van de CPU in Hz (clock cycles per seconde).

Indien je bovenstaande code gebruikt kunnen er een aantal problemen optreden. Ten eerste kan het zijn dat je start of stop meting niet op de correcte plaats zit vanwege *out of order execution* (hiervoor zou je een extra instructie kunnen bijplaatsen, b.v. `cpuid`, die de instructie stroom serializeert). Ten tweede kan je computer zijn clock snelheid dynamisch wijzigen (b.v. door powermanagement). Ten derde kan je problemen hebben op systemen met meerdere cores, waarbij de waardes van de TSC kunnen verschillen van core tot core en je niet kan garanderen dat je start en stop metingen op dezelfde core zullen doorgaan. Dit laatste zou je in Linux kunnen aanpakken door een oproep naar `sched_setaffinity` waarbij je er dan voor kan zorgen dat je code maar op één enkele core zal draaien. We gaan al deze mogelijke problemen negeren...

Om timings te doen zullen we gebruik maken van de klasse `Bench` in `bench.hpp`. Deze klasse verwacht het te timen stuk code a.d.h.v. een template argument. Je maakt een klasse aan voor je experiment die een `operator()()` heeft om het experiment te runnen en een methode `use_result` die een berekende waarde b.v. naar het scherm schrijft zodat je te timen code niet volledig zal worden weggeoptimaliseerd. Een voorbeeld:

```
#include <iostream>
#include <vector>
#include "bench.hpp"

const std::size_t N = 1 << 5; // known at compile time!

struct SumTest
{
```

```
    double sum;
    std::vector<double> a;

    SumTest() : a(N, -1) { }

    void operator()()
    {
        sum = 0;
        for(std::size_t i = 0; i < N; ++i)
            sum += a[i];
    }

    void use_result()
    {
        std::cout << sum << std::endl;
    }
};

int main()
{
    const size_t n = 50;      // number of experiments
    const size_t warmup = 5;  // number of warmup experiments for cache loads
    typedef TscTimer timer;   // use TSC timer
    Bench< SumTest, n, warmup, timer > b(std::cout);
    b.dump(std::cerr);        // output raw timing data to stderr
}
```

1. Compileer dit voorbeeld met `g++` en `-O0`.
2. Voer het programma uit als volgt: `./sum 2>sum-g++-O0 | tail`. De tijdsmetingen zijn nu bewaard in een bestand `sum-g++-O0`. Doe dit ook voor `-O3`. Plot deze bestanden m.b.v.

   ```
   echo 'plot "sum-g++-O0", "sum-g++-O3"' | gnuplot -persist
   ```

3. Bestudeer ook de output van de tijdsmeting op je scherm. Wat is de "`warmup`"? Heb je een goede waarde gekozen voor je test? Let op de verhouding `max/min`. Voor de metingen in je huistaak gebruik je enkel "goede" metingen! (Wat zijn "goede" metingen?) (Beweeg eens een window over het scherm tijdens je metingen en kijk naar de plot van de metingen...)

1.3 Expression templates

In de file `matrix_ops.hpp` vind je een klassieke object-georienteerd `operator+` voor het optellen van twee `Matrix` objecten:

```
template <typename Shape>
tws::Matrix<Shape> operator+(tws::Matrix<Shape> A, const tws::Matrix<Shape>& B)
{
    assert(A.rows() == B.rows() && A.cols() == B.cols());
    for(std::size_t i = 0; i < A.rows(); ++i)
```

```
        for(std::size_t j = 0; j < A.cols(); ++j)
            A(i, j) += B(i, j);
    return A;
}
```

Merk op dat we `A` "by value" doorgeven en we dus op een kopie werken. (Om het eenvoudig te houden kan deze code enkel matrices met dezelfde `Shape` optellen.)

Maak nu een bestand `matrixtest.cpp` dat we zullen gebruiken om de snelheidswinst van expression templates op te meten:

```
#include <iostream>
#include "bench.hpp"
#include "matrix.hpp"
#ifdef EXPR
#  include "expr.hpp"
#else
#  include "matrix_ops.hpp"
#endif

const std::size_t rows = 400;
const std::size_t cols = 300;
typedef tws::Matrix<tws::ColumnMajor<double>> Matrix;

struct MatrixTest
{
    Matrix A, B, C;

    MatrixTest() : A(rows, cols, -1), B(rows, cols, 2), C(rows, cols, 0) { }

    void operator()()
    {
        C = A + (B + (A + A + A) + B) + B;
    }

    void use_result()
    {
        std::cout << C(0,0) << ", " << C(rows-1, cols-1) << std::endl;
    }
};

// and a main routine as in the previous example
```

Je hebt volgende operator nog nodig in de klasse `Matrix`:

```
Matrix& operator=(const Matrix& A)
{
    assert((rows() == A.rows()) && (cols() == A.cols()));
    std::copy(begin(A.data_), end(A.data_), begin(this->data_));
    return *this;
}
```

Opmerking: voor het timen willen we uiteraard de `assert` statements uitschakelen. Dit doe je door de `-DNDEBUG` mee te geven aan de compiler!

Je kan nu de test draaien met de vlag `-DEXPR` om expressie templates te gebruiken, of zonder de vlag om de klassieke methode te gebruiken.

1. Bekijk de man page van `g++` eens: `man g++`. Je kan deze info ook online vinden. Nuttige optimalisatievlaggen zijn bv: `-march` en `-mtune`. Kijk ook eens naar wat de standaard optimalisatievlaggen `-O1`, `-O2`, `-O3`, `-Os` en `-Ofast`.

2. Time je code met expression templates voor verschillende compiler optimalisatievlaggen.

3. Time je code zonder expression templates voor verschillende compiler optimalisatievlaggen.

2 Opgave voor thuis

Belangrijk: Je draait alle timings op de machines in het labo.

1. Laat de compiler loop unrolling doen op het programma met `SumTest` door de optie `-funroll-loops`.

2. Maak een variant van het programma met `SumTest` om de som niet in één variabele te berekenen maar bijvoorbeeld in vier variabelen (hiervoor moet je zelf manueel de lus ook ontrollen). (Let erop dat je nog steeds het juiste resultaat berekent!) We verminderden nu de data afhankelijkheid van de berekeningen.

3. Zorg dat je ook vermenigvuldigingen kan doen in `MatrixTest` en kies dan zelf een uitdrukking met een aantal matrices waarin je + en * gebruikt als test in `MatrixTest`. Doe nu opnieuw timings met en zonder expression templates.

2.1 Indienen

Indienen doe je per email naar `gowri.suryanarayana@cs.kuleuven.be` ten laatste op donderdag Woensdag 16 December 2015 om 14u00:

1. Geef je aangepaste code van `operator()()` uit `SumTest`. Leg uit waarom dit beter zou moeten werken.

2. Voor `SumTest`: maak één nuttige grafiek van je timings voor verschillende optimalisaties. Minimum `-O3`, `-O3 -funroll-loops`, `-O3 -funroll-loops` voor je aangepaste variant, en een vrije keuze aan optimalisatievlaggen die jouw het beste resultaat gaf dat je testte. Deze grafiek is *niet* een gnuplot grafiek zoals in de oefenzitting. Je plot het gemiddelde (of de mediaan) en eventueel de spread (een zogenaamde boxplot). Dit kan je b.v. doen m.b.v. gnuplot of Matlab. Je plot niet de clock ticks maar rekent dit om naar ms.

3. Maak een gelijkaardige grafiek voor `MatrixTest` met de uitdrukking die + en * gebruikt. Zet de uitdrukking in de titel van de grafiek. Ook hier gebruik je de tijd in ms. Zorg dat het duidelijk is in de legende welke timings met expression templates zijn en welke zonder.

De stukken code en je commentaar zet je als plain text in je email, grafieken als pdf of als png. Als onderwerp van je email gebruik je "`TWS C++ 3`". Vermeld in je email ook hoeveel tijd je aan deze huistaak besteedde en eventuele problemen die zich voordeden.