Politechnika Wrocławska

Wydział Elektroniki

Organizacja i Architektura Komputerów

Automatyczna wektoryzacja kodu z wykorzystaniem GCC

Autor:
Jakub Sokołowski 226080

Prowadzący:
Dr inż. Tadeusz Tomczak

Grupa:
CZ-TP-11

23 maja 2019

Spis treści

1 Wstęp

1.1 Wektoryzacja

W informatyce, wektoryzacja to proces konwertowania skalarnej implementacji algorytmu, który wykonuje operację na jednej parze operandów, na implementację wektorową gdzie pojedyncza operacja można wykonać na wielu parach operandów jednocześnie. Instrukcje, umożliwiajace takie operacje to tzw. instrukcje *SIMD* (*Single Instruction Multiple Data*)

Intel wprowadził 80-bitowe rejestry wektorowe i zestaw instukcji *SIMD* (nazwany *MMX*) w swoich procesorach w 1997 roku. W późniejszych latach, wprowadzone zostały zestawy instrukcji *SSE* dla rejestrów 128-bitowych i *AVX* dla rejestrów 256-bitowych [1].

Chociaż zestawy instrukcji *SIMD* istnieją już od długiego (w skali rozwoju procesorów) czasu, dostęp do tych instrukcji był możliwy wyłącznie z poziomu asemblera, lub poprzez przeplatanie wysokopoziomowego kodu *C* i *C++* z "prawie-asemblerowym" kodem [2]. Obecnie, nowoczesne kompilatory takie jak *Clang* i *GCC* mają możliwość przekształcania zwykłego kodu źródłowego napisanego w *C* lub *C++* na operacje wektorowe - proces ten to **automatyczna wektoryzacja**.

1.2 Cel projektu

Projekt ma za zadanie zbadać jak można automatycznie zwektoryzować kod z wykorzystaniem kompilatora GCC oraz jakie są wymagania i ograniczenia automatycznej wektoryzacji.

2 Sposób przeprowadzenia badań

2.1 Metoda badań

Fragmentu kodu badane w tym projekcie zostały skompilowane przy użyciu wspólnych flag optymalizacji, a przez analizę raportów kompilatora i wynikowego kodu asemblera zbadano czy wektoryzacja zaszła.

2.2 Sprzęt i oprogramowanie

```
System    : Ubuntu 18.04.2 LTS
Kernel    : 4.15.0-50-generic
Procesor  : Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz
Kompilator: gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04)
```

2.3 Flagi optymalizacji

Kod źródłowy przykładów został skompilowany z użyciem następujących flag:

-03 - włącza szereg optymalizacji, w tym wektoryzacje

-ffast-math - umożliwia wektoryzacje operacji matematycznych

-fopt-info-vec - generuje raporty dotyczące wektoryzacji kodu

W niektórych przykładach użyto innych flag optymalizacji, lub nie użyto żadnej - zmiana flag będzie wyraźnie podkreślona w każdym takim przypadku.

3 Automatyczna wektoryzacja i jej ograniczenia

3.1 Kod który można zwektoryzować

3.1.1 Proste operacje

W celu rozdzielenia obliczeń do jednostek wektorowych procesora kompilator musi dokładnie przeanalizować zależności i efekty uboczne kodu źródłowego. Pierwszym etapem tego procesu jest wykrywanie sekcji kodu, w których można zastosować instrukcje *SIMD*. Najprostszym przykładem takiej sekcji jest pętla, która wykonuje pewne obliczenia na tablicach [3]. Przykład takiego kodu znajduje się na lisitngu 1. Fragment kodu asemblera który wygenerował kompilator oraz fragment raportu wektoryzacji znajdują się odpowiednio na listingach 2 i 3.

Listing 1: Prosta pętla

```
int a[256], b[256], c[256];
void loop () {
    int i;
    for (i=0; i<256; i++){
        a[i] = b[i] + c[i];
    }
}
int main() {
    loop();
}
```

Listing 2: Fragment kodu asemblera

```
.L2:
        movdqa (%rcx,%rax), %xmm0
        paddd (%rdx,%rax), %xmm0
        movaps %xmm0, (%rsi,%rax)
        addq $16, %rax
        cmpq $1024, %rax
        jne .L2
        rep ret

...
```

Listing 3: Fragment raportu wektoryzacji kompilatora

```
./src/loop-examples/loop_basic_simple.c:5:5: note: loop vectorized
```

Na listingu 2 widać wyraźnie efekty automatycznej wektoryzacji - sumowanie tablic w pętli zostało zawarte w 3 instrukcjach *SIMD*. Dla porównania, na listingu 4 znajduje się wynikowy kod asemblera tego samego przykładu, ale skompilowany bez użycia flagi *-O3* - widać, że wektoryzacja nie zaszła.

Listing 4: Niezoptymalizowany kod

```
.L4:
        movl -4(%rbp), %eax
        cltq
        leaq 0(,%rax,4), %rdx
        leaq c(%rip), %rax
        movl (%rdx,%rax), %edx
        movl -8(%rbp), %eax
        cltq
        leaq 0(,%rax,4), %rcx
```

```
        leaq b(%rip), %rax
        movl (%rcx,%rax), %eax
        cmpl %eax, %edx
        cmovl %eax, %edx
        movl -8(%rbp), %eax
        cltq
        leaq 0(,%rax,4), %rcx
        leaq a(%rip), %rax
        movl %edx, (%rcx,%rax)
        addl $1, -4(%rbp)
.L3:
        cmpl $65535, -4(%rbp)
        jle .L4
```

3.1.2 Zagnieżdżone pętle

Oprócz zwykłych pętli, *GCC* jest w stanie automatycznie zwektoryzować pętle zawarte w pętlach czyli tzw. zagnieżdżone pętle.

Listing 5: Zagnieżdżona pętla

```
#define N (1L << 16)
double a[N], b[N];
void loop() {
    int i = 0;
    int j = 0;

    for (j = 0; j < N; j++) {
        for (i = 0; i < N; i++) {
            a[i + j * N] += b[i + j * N];
        }
    }
}

int main() {
    loop();
}
```

Listing 6: Fragment kodu asemblera

```
...
.L3:
        movapd (%rdx,%rax), %xmm0
        addpd (%rcx,%rax), %xmm0
        movaps %xmm0, (%rdx,%rax)
        addq $16, %rax
        cmpq $524288, %rax
        jne .L3

...
```

Listing 7: Fragment raportu wektoryzacji kompilatora

```
./src/loop-examples/nested_loop_simple.c:11:9: note: loop vectorized
./src/loop-examples/nested_loop_simple.c:11:9: note: loop versioned for vectorization
    ↪ because of possible aliasing
```

Poza zagnieżdżonymi pętlami, *GCC* wspiera różne warianty pętli- pętle *for* i *while*, pętle odwrotne, pętle ograniczone wskaźnikami i indeksami, pętle na tablicach wielowymiarowych oraz wiele innych. Kompletną listę wspieranych pętli można znaleźć na stronie kompilatora *GCC* [3].

3.1.3 Funkcje matematyczne w pętlach

Kompilator *GCC*, oprócz podstawowych działań takich jak dodawania czy mnożenie, potrafi zwektoryzować funkcje matematyczne takie jak *sin()* czy *log()*. Generalnie, funkcje które mogą być wstawione "w linii" (w miejsce wywołania funkcji nie jest wstawiany wskaźnik do funkcji tylko cały kod funkcji) mogą zostać zwektoryzowane. Wektoryzację operacji matematycznych w *GCC* zapewnia biblioteka *libmvec*. Na listingu 9 widać, że została wywołana funkcja *_ZGVbN4v_sinf* - kod został zwektoryzowany. Do wykorzystania *libmvec* konieczna jest flaga *-ffast-math*.

Listing 8: Funkcja sinus "w linii"

```
#define N (65536)
#include <math.h>
void loop(float a[], float b[], float c[]) {
    for (int i=0; i<N; i++)
        a[i] += sinf(b[i] * c[i]);
}

int main() {
    float a[N], b[N], c[N];
    loop(a,b,c);
}
```

Listing 9: Fragment kodu asemblera

```
...
  .L2:
    movq %rdi, 16(%rsp)
    movq %rax, 8(%rsp)
    addq $16, %r15
    movups (%rax), %xmm1
    mulps %xmm1, %xmm0
    call _ZGVbN4v_sinf


...
```

Listing 10: Fragment raportu wektoryzacji kompilatora

```
./src/loop-examples/loop_sin_opt_simple.c:8:5: note: loop vectorized
./src/loop-examples/loop_sin_opt_simple.c:8:5: note: loop versioned for vectorization
    ↪ because of possible aliasing
```

3.2 Niewyrównane, dynamiczne dane a wektoryzacja

We wszystkich powyższych przykładach, operacje w pętli były wykonywana na tablicach o rozmiarze znanym w czasie kompilacji. Jak dzieje się w przypadku tablic dynamicznych ?

Listing 11: Tablice Dynamiczne

```
#define SIZE (1L << 16)

void loop(double *a, double *b) {
    int i;

    for (i = 0; i < SIZE; i++) {
        a[i] += b[i];
    }
}

int main() {
    double a[SIZE];
    double b[SIZE];
    loop(a,b);
}
```

Listing 12: Optymalny kod asemblera

```
...
.L2:
        movapd (%rdi,%rax), %xmm0
        addpd (%rsi,%rax), %xmm0
        movaps %xmm0, (%rdi,%rax)
        addq $16, %rax
        cmpq $524288, %rax
        jne .L2




...
```

Listing 13: Generowane opcje

```
.LFB10:
        .cfi_startproc
        leaq 16(%rsi), %rax
        cmpq %rax, %rdi
        jnb .L10
        leaq 16(%rdi), %rax
        cmpq %rax, %rsi
        jb .L8
...
.L5:
        movupd (%r9,%rdx), %xmm0
        addl $1, %ecx
        addpd (%r8,%rdx), %xmm0
        movaps %xmm0, (%r8,%rdx)
        addq $16, %rdx
        cmpl %r10d, %ecx
        jb .L5
...
.L8:
        xorl %eax, %eax
        .p2align 4,,10
        .p2align 3
.L2:
        movsd (%rdi,%rax), %xmm0
        addsd (%rsi,%rax), %xmm0
        movsd %xmm0, (%rdi,%rax)
        addq $8, %rax
        cmpq $524288, %rax
        jne .L2
```

Listing 14: Raport wektoryzacji dla listingów 15 i 11

```
./src/loop-examples/loop1_opt_simple.c:9:5: note: loop vectorized
./src/loop-examples/loop1_opt_simple.c:9:5: note: loop versioned for vectorization
    ↪ because of possible aliasing
```

Listing 15: Tablice dynamiczne - optymalny kod

```
#define SIZE (1L << 16)

void loop(double * restrict a, double * restrict b) {
    int i;

    double *x = __builtin_assume_aligned(a, 16);
    double *y = __builtin_assume_aligned(b, 16);

    for (i = 0; i < SIZE; i++) {
        x[i] += y[i];
    }
}

int main() {
    double a[SIZE];
    double b[SIZE];
    loop(a,b);
}
```

Pierwszym krokiem jaki wykonuje *GCC* jest sprawdzenie, czy adresy dynamicznych tablic nachodzą na siebie (listing 13 etykieta .LFB10). Jeśli nachodzą, dodawanie wykonywane jest liczba po liczbie za pomocą instrukcji *addsd* (etykieta *L2*, jeśli nie, dodawanie jest zwektoryzowane i odbywa się za pomocą instrukcji *addpd* (etykieta *L5*).
Dzieje się tak, ponieważ kompilator ma ograniczone informacje o tablicach. Do skutecznej wektoryzacji *GCC* potrzebuje informacji, że dane w tablicach się nie nie nakładają oraz, że dane są wyrównane do rozmiaru rejestru *SSE* (128 bitów - 16 bajtów). Pierwszą informację można przekazać poprzez słowo kluczowe *restricted*, a drugą poprzez wbudowaną funkcję *__builtin_assume_alligned()*. Zoptymalizowany z wykorzystaniem słów kluczowych kod znajduje się na listingu 15, a jego kod wynikowy asemblera na listingu 12.

3.3 Kod którego nie można zwektoryzować

3.3.1 Skomplikowana instrukcje warunkowe

Ponieważ instrukcji *SIMD* wykonują te same operacje na tych samych danych, nie jest możliwe by różne iteracje pętli wykonywały różne instrukcje - nie mogą występować rozgałęzienia. Instrukcje warunkowe mogą być zwektoryzowane jedynie wtedy, gdy mogą być zaimplementowane jako zamaskowane przypisania (*masked assignment*) [6].

Listing 16: Instrukcje warunkowe

```
#define N (65536)

int a[N], b[N], c[N];

int loop () {
    int i, j;
    for (i=0; i<N; i++){
        if(a[i] > b[i] * c[i])
            a[i] += b[i] > c[j] ? b[i] : c[j];
        else
            a[i] = 23;
    }
    return a[0];
}

int main() {
    int a[N], b[N], c[N];
    loop(a,b,c);
}
```

Listing 17: Fragment kodu asemblera

```
...
 .L2:
        movl (%r10,%rax), %edx
        movl (%rsi,%rax), %r8d
        movl (%rdi,%rax), %ecx
        imull %edx, %r8d
        cmpl %r8d, %ecx
        jle .L3
        cmpl %edx, (%rsi,%r9,4)
        cmovge (%rsi,%r9,4), %edx
        addl %ecx, %edx
        movl %edx, (%rdi,%rax)
        addq $4, %rax
        cmpq $262144, %rax
        jne .L2



...
```

Listing 18: Fragment raportu wektoryzacji kompilatora

```
./src/loop-examples/control_flow.c:8:5: note: not vectorized: control flow in loop.
./src/loop-examples/control_flow.c:8:5: note: bad loop form.
```

3.3.2 Zależność danych

Warunkiem koniecznym równoległego przetwarzania danych w tablicy jest niezależność tych danych. Jeśli wyniki następnej iteracji zależy od wyniku porzedniej, kodu nie da się zwektoryzować.

Listing 19: Zależność "Read after Write"

```
#define N (65536)
int a[N];

void loop () {
    int j;

    a[0]=0;
    for (j=1; j<N; j++)
        a[j]=a[j-1]+1;
}

int main() {
    loop();
}
```

Listing 20: Fragment kodu asemblera

```
...
.L2:
        movl %edx, (%rax)
        addq $4, %rax
        addl $1, %edx
        cmpq %rcx, %rax
        jne .L2
        rep ret

...
```

Listing 21: Fragment raportu wektoryzacji kompilatora

```
./src/loop-examples/read_after_write_simple.c:8:5: note: bad data dependence.
./src/loop-examples/read_after_write_simple.c:8:5: note: not vectorized, possible
    ↪ dependence between data-refs
```

3.3.3 Niewspierane operacje matematyczne

Nie wszystkie operacje matematyczne mogą być zwektoryzowane - przykładem operacji która nie może jest operacja modulo na zmiennych zmiennoprzecinkowych. Operacja ta nie może być zwektoryzowana, ponieważ *GCC*, w przeciwieństwie do funkcji *sin*() czy *log()*, nie posiada wbudowanych funkcji do tej operacji, i nie może zamienić jej na pojedyncze instrukcje.

Listing 22: Operacja Modulo

```
#include <stdlib.h>
#include <math.h>

#define N (65536)

void loop(float a[], float b[], float c[]) {
    for (int i=0; i<N; i++)
        a[i] += fmod(b[i], c[i]);
}

int main() {
    float a[N], b[N], c[N];
    loop(a,b,c);
}

...
```

Listing 23: Fragment kodu asemblera

```
...
.L3:
        flds (%rdx,%rcx)
        flds (%rsi,%rcx)
.L2:
        fprem
        fnstsw %ax
        testb $4, %ah
        jne .L2
        fstp %st(1)
        pxor %xmm0, %xmm0
        pxor %xmm2, %xmm2
        fstpl -8(%rsp)
        cvtss2sd (%rdi,%rcx), %xmm0
        addsd -8(%rsp), %xmm0
        cvtsd2ss %xmm0, %xmm2
        movss %xmm2, (%rdi,%rcx)
        addq $4, %rcx
        cmpq $262144, %rcx
        jne .L3
        rep ret
...
```

Listing 24: Fragment raportu wektoryzacji kompilatora

```
./src/loop-examples/fmod_simple.c:8:5: note: function is not vectorizable.
./src/loop-examples/fmod_simple.c:8:5: note: not vectorized: relevant stmt not
    ↪ supported: _12 = fmod (_11, _8);
```

4 Praktyczna wektoryzacja

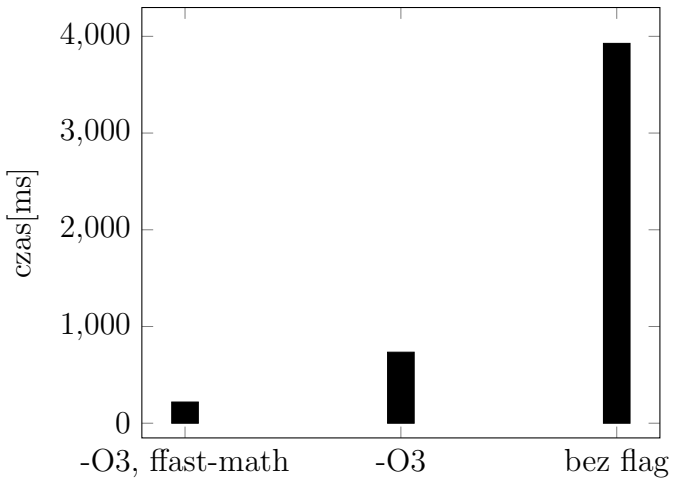
4.1 Fraktal Mandelbrota

Fraktal Mandelbrota to zbiór liczb zespolonych c dla których funkcja:

$$f_c(z) = z^2 + c$$

iterując od $z = 0$ nie jest rozbieżna.

Z punktu widzenia programu, dla każdego piksela iterujemy po wzorze $z_{n+1} = z_n^2 + c$ do momentu gdy $|z_n| > 2$ lub maksymalna liczba iteracji zostanie przekroczona. Ponieważ przynależność każdego piksela do zbioru można policzyć niezależnie, a funkcja $f_c(z) = z^2 + c$ może zostać "wstawiona w linii", algorytm nadaj się do automatycznej wektoryzacji. Na wykresie 1 znajdują uśrednione wyniki 1000 pomiarów generowania fraktalu o wymiarach 800x800 i maksymalnej liczbie iteracji wynoszącej 500.



Rysunek 1: Porównanie wpływu ustawionych flag na czas generowania Fraktala

Po wygenerowaniu, fraktal jest zapisywany w formacie BMP i za pomocą biblioteki *SDL* można go wyświetlić. Na rysunku 2 przedstawiony został fraktal wygenerowany podczas pomiarów.

Rysunek 2: Przykładowy wygenerowany fraktal

5 Wnioski

Z wersji na wersję, *GCC* poszerza swoje wsparcie dla automatycznej wektoryzacji. Wiele przykładów kodu, których nie można było automatycznie zwektoryzować w *GCC 4* czy *GCC 5*, w wersji *GCC 7.4.0* można łatwo zoptymalizować. Rozwijane są również narzędzie wspomagające programistów w pisaniu kodu który można zwektoryzować - w najnowszej wersji *GCC* - 9, raporty kompilatora z optymalizacji generowane przy użyciu flagi -fopt-info są znacznie bardziej przejrzyste[10]. Działania jakie trzeba podjąć by móc otrzymać automatycznie zwektoryzowany kod można podsumować następująco:

- Używaj najnowszej wersji kompilatora
- Używaj tablic o rozmiarach znanych w czasie kompilacji
- Jeśli używasz tablic dynamicznych, przekaż odpowiednie informacje kompilatorowi
- Dostęp do tablic przez indeksy
- Ogranicz instrukcje warunkowe w pętli
- Ogranicz wywołania funkcji w pętli, jeśli są konieczne, upewnij się że można je "wstawić w linii"
- Kieruj się wskazówkami z raportów optymalizacji kompilatora

Literatura

[1] Instrukcje SIMD w procesorach Intel [dostęp 16-05-2019]
http://tm.spbstu.ru/images/d/db/Intel_simd.pdf

[2] Wbudowane funkcje do wektoryzacji GCC [dostęp 16-05-2019]
https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html

[3] Przykłady automatycznej wektoryzacji pętli w GCC[dostęp 16-05-2019]
https://gcc.gnu.org/projects/tree-ssa/vectorization.html

[4] Opcje optymalizacji GCC [dostęp 16-05-2019]
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[5] Funkcje matematycznych w libmvec [dostęp 16-05-2019]
https://github.com/lattera/glibc/blob/
master/sysdeps/unix/sysv/linux/x86_64/libmvec.abilist

[6] Maskowane przypisania w procesorach Intel [str. 125]
"Intel Xeon Phi Coprocessor High Performance Programming"
James Jeffers, James Reinders, 2013

[7] Software optimization resources [dostęp 16-05-2019]
https://www.agner.org/optimize/

[8] A practical guide to SSE SIMD with C++ [dostęp 16-05-2019]
http://sci.tuomastonteri.fi/programming/sse

[9] What to do when auto-vectorization fails? [dostęp 16-05-2019]
https://software.intel.com/en-us/articles/what-to-do-when-auto-vectorization-fails

[10] GCC 9 Release Series Changes, New Features, and Fixes [dostęp 16-05-2019]
https://gcc.gnu.org/gcc-9/changes.html