Векторные инструкции процессора

Кирилл Батузов

ИСП РАН

16 октября 2017

Введение

При разработке процессора производители руководствуются несколькими целями:

- высокая производительность,
- низкое энергопотребление,
- низкая стоимость.

Мы будем говорить сегодня преимущественно о высокопроизводительных процессорах.

- Быстрее.
- Выше (производительность).
- Мощнее.

Введение

- Существует предел, до которого можно поднимать частоту процессора.
- Этот лимит близок к исчерпанию.
- Дальнейшее увеличение производительности возможно за счет выполнения нескольких действий параллельно.
  - Многоядерные процессоры.
  - Конвейер внутри процессора.
  - Суперскалярные архитектуры.
  - Векторные инструкции!

Все примеры будут рассматриваться на двух архитектурах: x86_64 и ARM.

- Архитектура x86_64 вам знакома по курсу «Архитектура ЭВМ и язык ассемблера».
  - Будем использовать синтаксис AT&T, так что не забывайте писать доллары, процентики и аргументы в другом порядке.
- Архитектура ARM
  - 16 регистров общего назначения: `r0`, `r1`, ..., `r15`.
  - `r13` — указатель стека, `r14` — адрес возврата, `r15` — счетчик команд.
  - Трехадресная архитектура, результат идет первым:
    - `add r0, r1, r2`.
  - Архитектура с явными загрузками и сохранениями.
    - Инструкции оперируют с регистрами и константами, к памяти обращаются только специально выделенные инструкции `ldr` и `str`.

Векторные инструкции

Векторные инструкции

- Служат для обработки массивов данных.
- Выполняют одну и ту же операцию над несколькими элементами массива.
  - SIMD — Single Instruction Multiple Data.
- Имеют фиксированную битовую длину.
  - Ближе к математическим векторам.
  - Никак не связаны со структурой данных «vector», хотя и могут использоваться в ее реализации.
- Присутствуют в различных процессорных архитектурах.

Векторные регистры

- Имеют фиксированную битовую длину.
- Типичные длины векторов — 64, 128, 256, 512 бит.
- Интерпретация вектора зависит от инструкции, которая с ним работает.
  - Например, 128-битный вектор может быть вектором из 4-х 32-битных чисел, или 16-ти 8-битных.
- Архитектура x86_64 имеет 64-битные `%mm` регистры, 128-битные `%xmm` регистры, 256-битные `%ymm` регистры, 512-битные `%zmm` регистры.
- Архитектура ARM имеет 64-битные `d` регистры и 128-битные `q` регистры.

Примеры векторных инструкций

Сложение векторов из 4-х 32-битных чисел

x86_64
```
paddd %xmm1, %xmm0
```
ARM
```
vadd.i32 q0, q0, q1
```

Поэлементный максимум векторов из 8-ми 16-битных знаковых чисел

x86_64
```
pmaxsw %xmm1, %xmm0
```
ARM
```
vmax.s16 q0, q0, q1
```

Пересылка между векторным регистром и памятью

x86_64
```
movdqu %xmm0, (%ebx)
```
ARM
```
vst1.32 {d0, d1}, [r0]
```

Краткая история SIMD расширений x86_64

- MMX — 8 64-битных регистров.
- SSE — 8 128-битных регистров, только 4xfloat32 инструкции.
- SSE2 — добавлены 2xfloat64, 4xint32 и другие инструкции; +8 регистров в AMD64.
- SSE3, SSSE3, SSE4 (SSE4.1, SSE4.2, SSE4a).
- AVX — 256-битные регистры, только операции с плавающей точкой, +16 128-битных регистров.
- AVX2 — целочисленные операции.
- AVX512.

Использование векторных инструкций

- Написание соответствующего ассемблера вручную.
- Использование компиляторных интринсиков.
- Использование компиляторных «обобщенных векторов».
- Автоматическая векторизация.
- Высокоуровневые языки с поддержкой векторных типов?

Написание векторных инструкций вручную

Написание векторных инструкций вручную

- Ассемблерные вставки?
  - НЕТ!
- Написание отдельных функций на ассемблере и вызов их из основного кода на высокоуровневом языке программирования?
  - Можно.

Плюсы написания вручную.

- Можно использовать необычные инструкции, для которых нет никаких аналогов в языках высокого уровня.
  - `VUZP.i32` / `punpckldq`.

Минусы написания вручную.

- Стандартный набор минусов написания ассемблера вручную.

Пример из реальной жизни: x264.

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

```
foo:
    lea     a(%rip), %r8
    lea     b(%rip), %r9
    lea     c(%rip), %r10
    xor     %rcx, %rcx
.loop:
    movdqu  (%r8, %rcx), %xmm0
    paddd   (%r9, %rcx), %xmm0
    movdqu  %xmm0, (%r10, %rcx)
    add     $0x10, %rcx
    cmp     $0x400, %rcx
    jl      .loop
    ret
```

```
foo:
    movw        r1, #:lower16:a
    movt        r1, #:upper16:a
    movw        r2, #:lower16:b
    movt        r2, #:upper16:b
    movw        r3, #:lower16:c
    movt        r3, #:upper16:c
    add         r0, r1, #0x400
.loop:
    vld1.32     {d0-d1}, [r1]
    vld1.32     {d2-d3}, [r2]
    vadd.i32    q0, q0, q1
    vst1.32     {d0-d1}, [r3]
    add         r1, r1, #0x10
    add         r2, r2, #0x10
    add         r3, r3, #0x10
    cmp         r0, r1
    bne         .loop
    mov         r15, r14
```

Использование компиляторных интринсиков

- Intrinsics — встроенные «функции», предоставляемые компилятором.
- С точки зрения языка Си они имеют синтаксис функций.
- Компилятор их знает и обрабатывает специальным образом.
  - Например, он может сгенерировать эквивалентную векторную инструкцию вместо вызова функции.
- В конечном итоге, intrinsics — это способ расширить язык Си дополнительными возможностями, не меняя его синтаксис.
- Набор и смысл intrinsics — архитектурно зависимы.
- Разработчики процессора стараются сделать одинаковый набор intrinsics в различных компиляторах под свою архитектуру.
- Есть переносимость между компиляторами, но нет переносимости между архитектурами.

```
#include <emmintrin.h>
int a[256], b[256], c[256];
void foo(void) {
    __m128i *va = (void *)a, *vb = (void *)b,
            *vc = (void *)c;
    for (int i = 0; i < 256 / 4; i++)
        vc[i] = _mm_add_epi32(va[i], vb[i]);
}
```

```
foo:
    <...>
    .p2align 4,,10
    .p2align 3
.L2:
    movdqa  (%rcx,%rax), %xmm0
    paddd   (%rdx,%rax), %xmm0
    movaps  %xmm0, (%rsi,%rax)
    addq    $16, %rax
    cmpq    $1024, %rax
    jne     .L2
    rep ret
```

```
#include <arm_neon.h>
int a[256], b[256], c[256];
void foo(void) {
    uint32x4_t *va = (void *)a, *vb = (void *)b,
               *vc = (void *)c;
    for (int i = 0; i < 256 / 4; i++)
        vc[i] = vaddq_u32(va[i], vb[i]);
}
```

```
    <...>
.L2:
    vld1.32     {q8}, [r3]
    add         r3, r3, #16
    cmp         r3, r0
    vld1.32     {q9}, [r1]
    add         r1, r1, #16
    vadd.i32    q8, q8, q9
    vst1.32     {q8}, [r2]
    add         r2, r2, #16
    bne         .L2
    bx          lr
```

Использование компиляторных обобщенных векторов

- Еще один способ расширить язык Си в компиляторе — атрибуты типов и переменных.
- Атрибут `vector_size` позволяет объявить тип данных, являющийся вектором нескольких базовых типов.
- Специфичный для `gcc` атрибут.
- Работает с разными архитектурами.
- Есть переносимость между архитектурами, но нет переносимости между компиляторами.
- Что будет, если требуемый вектор не соответствует поддерживаемому в архитектуре?
  - Ничего хорошего.
  - Код будет работать корректно.
  - Производительность может очень сильно пострадать.

```
int a[256], b[256], c[256];
typedef int i32x4 __attribute__((vector_size(16)));
void foo(void) {
    i32x4 *va = (void *)a, *vb = (void *)b,
          *vc = (void *)c;
    for (int i = 0; i < 256 / 4; i++)
        vc[i] = va[i] + vb[i];
}
```

```
foo:
    <...>
    .p2align 4,,10
    .p2align 3
.L2:
    movdqa  (%rcx,%rax), %xmm0
    paddd   (%rdx,%rax), %xmm0
    movaps  %xmm0, (%rsi,%rax)
    addq    $16, %rax
    cmpq    $1024, %rax
    jne     .L2
    rep ret
```

```
int a[256], b[256], c[256];
typedef int i32x4 __attribute__((vector_size(16)));
void foo(void) {
    i32x4 *va = (void *)a, *vb = (void *)b,
          *vc = (void *)c;
    for (int i = 0; i < 256 / 4; i++)
        vc[i] = va[i] + vb[i];
}
```

```
    <...>
.L2:
    vld1.32     {q8}, [r3]
    add         r3, r3, #16
    cmp         r3, r0
    vld1.32     {q9}, [r1]
    add         r1, r1, #16
    vadd.i32    q8, q8, q9
    vst1.32     {q8}, [r2]
    add         r2, r2, #16
    bne         .L2
    bx          lr
```

Автоматическая векторизация

Автоматическая векторизация

- Одна из оптимизаций в компиляторе.
- Работает с циклами.
- Необходимо «разрешить» компилятору использовать векторные инструкции.
  - `-msse4.1` для `x86_64`.
  - `-cpu=cortex-a8 -mfpu=neon -mfloat-abi=softfp` для ARM.
- Сказать компилятору векторизовать циклы.
  - `-ftree-vectorize`.
  - Данная опция включена в `-O3`.

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

```
arm-unknown-linux-gnueabi-gcc -O2 -ftree-vectorize
-mfpu=neon -mfloat-abi=softfp -mcpu=cortex-a8 -S
```

```
    <...>
.L2:
    vld1.32     {q8}, [r3]
    add         r3, r3, #16
    cmp         r3, r0
    vld1.32     {q9}, [r1]
    add         r1, r1, #16
    vadd.i32    q8, q8, q9
    vst1.32     {q8}, [r2]
    add         r2, r2, #16
    bne         .L2
    bx          lr
```

gcc -O2 -ftree-vectorize -msse4.1 -S

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

gcc -O2 -ftree-vectorize -mavx512f -S

```
.L2:
    vmovdqu64   (%rcx,%rax), %zmm0
    vpaddd      (%rsi,%rax), %zmm0, %zmm0
    vmovdqu32   %zmm0, (%rdx,%rax)
    addq        $64, %rax
    cmpq        $1024, %rax
    jne         .L2
    rep ret
```

Сложности при использовании векторных инструкций

- Зависимости по данным между итерациями цикла.
  - Переписать цикл по-другому.

```
void foo(int *a) {
    for (int i = 1; i < 256; i++)
        a[i] += a[i - 1];
}
```

- Неизвестное или некратное размеру вектора число итераций цикла.
  - Loop peeling.

```
void foo(int *a, int x, int n) {
    for (int i = 0; i < n; i++)
        a[i] *= x;
}
```

- Неизвестные зависимости по данным между итерациями цикла из-за возможного перекрытия массивов.
  - Loop versioning.
  - Ключевое слово `restrict`.

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

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

- Отсутствие нужных векторных инструкций.
  - ... взять другой процессор?

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

- Смешение разных типов.
  - Избегать таких ситуаций.

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

- Неизвестное или неправильное выравнивание.
  - Loop peeling, loop versioning.
  - Потеря производительности из-за невыровненного доступа.

```
int a[258], b[258], c[258];
void foo(void) {
    for (int i = 0; i < 256; i++)
        c[i] = a[i + 1] + b[i + 2];
}
```

- Зависимости по управлению.
  - Объединение по маске.

```
int a[256], b[256], c[256];
void foo(void) {
    for (int i = 0; i < 256; i++)
        if (a[i] % 2)
            c[i] = a[i] + b[i];
        else
            a[i] = a[i] - b[i];
}
```

Заключение

Подведем итог

Сегодня мы посмотрели

- что такое векторные инструкции векторные регистры,
- как их использовать в программе на языке Си,
- какие при этом могут возникать сложности.

Источники знаний

Векторные инструкции.

- ARM Architecture Reference Manual. ARM v7-A and ARM v7-R edition.
- AMD64 Architecture Programmer's Manual. Volume 4: 128-Bit and 256-Bit Media Instructions.
- Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2: Instruction Set Reference, A–Z.

Обобщенные вектора и автовекторизация.

- Using the GNU Compiler Collection (GCC): Vector Extensions.
  `https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html`
- Auto-vectorization in GCC.
  `https://gcc.gnu.org/projects/tree-ssa/vectorization.html`

Источники знаний

Интринсики.

- Intel Intrinsics Guide.

  `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`

- Summary of NEON intrinsics — ARM Infocenter.

  `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0491h/CIHJBEFE.html`

- ARM NEON Intrinsics — Using the GNU Compiler collection (GCC).

  `https://gcc.gnu.org/onlinedocs/gcc-4.8.0/gcc/ARM-NEON-Intrinsics.html`