SIMD

По классификации Майка Флинна SIMD, введённой в 1966 г., это:

Single Instruction (Stream), Multiple Data – один поток команд, множественный поток данных

SIMD вычислительная система содержит управляющий модуль (контроллер) и процессорные элементы – модули обработки данных, у каждого процессорного элемента может быть собственная память для хранения данных

Примеры: ряд исторических архитектур суперкомпьютеров(ILLIAC IV, ICL Distributed Array Processor (DAP) и ряд других), большинство современных GPU

Также есть SIMD-расширения различных архитектур (используется принцип, но нет всех атрибутов полноценной SIMD вычислительной системы)

SIMD-расширения - основы

- Современные процессоры имеют в своём арсенале векторные инструкции – инструкции, работающие с одномерными массивами данных (которые называются векторами)
- Позволяют сделать одну операцию над большим количеством данных за один раз – тем самым многократно повышая производительность, при наличии соответствующих условий
- Повышают производительность в широком спектре задач (обработка мультимедийных данных, научные и инженерные вычисления, любые задачи, которые поддаются векторизации)…
- …ценой увеличения сложности процессора, усложнения компилятора и необходимости, в ряде случаев, дополнительных действий со стороны программиста

SIMD-расширения в различных архитектурах

x86 и x86-64:

- MMX (Multimedia Extension) – 1997 г., первое расширение (только целочисленное, ряд ограничений)
- 3DNow! – 1998 г.(нынче устаревшее расширение, от компании AMD) – уже с поддержкой чисел с плавающей запятой
- SSE (Streaming SIMD Extensions) – первая версия в 1999 г., далее SSE2 (2000 г.), SSE3 (2004 г.), SSSE3 (2006 г.), SSE4 (2006 г.), SSE4a, SSE4.1, SSE4.2,SSE5 (предложен компанией AMD, не получил распространения)
- Advanced Vector Extensions (AVX) (2008 г.), AVX2 (2013 г.), AVX512 (2017 г.)

SIMD-расширения в различных архитектурах

ARM:

- SIMD, NEON, SVE, SVE2

MIPS:

- MDMX, MIPS-3D

SPARC:

- VIS, VIS2

RISC-V:

- P-extension

и другие…..

MMX

- 8 регистров по 64 бита (mm0 – mm7)
- каждый регистр - может содержать 2 32-битных, 4 16-битных или 8 8-битных целых числа (или одно значение в 64 бита, это времена когда основная архитектура была 32-битная)

Можно выделить группы инструкций:

- инструкции пересылки данных (позволяющих перемещать данные между регистрами MMX и целочисленными регистрами процессора или памятью)
- инструкции упаковки и распаковки данных (позволяющих преобразовывать элементы векторов в элементы с меньшей разрядностью или попарно объединять элементы с образованием элементов большей разрядности)

MMX – группы инструкций(продолжение)

- арифметические инструкции
- инструкции сравнения векторов с записью результата сравнения в виде битовой маски
- логические инструкции
- инструкции сдвиговые
- инструкции управления состоянием

SSE

- 16 дополнительных 128-битных регистров (в ранних версиях 8)

Название – от xmm0 до xmm15

- Управляющий регистр mxcsr
- Скалярные данные (просто одно значение на 128 бит) или векторизованные (упакованные) данные (scalar data или packed data)
- В каждом регистре могут содержаться 2 или 4 значения (64 или 32 бита) с плавающей запятой или 4, 8 или 16 целочисленных значений (на 32, 16 или 8 бит, соответственно)
- Расширен набор команд

Векторизация – пример ситуации

Пример – когда векторные инструкции дают выигрыш в производительности:

```
float A[ SIZE ] , B [ SIZE ] ;
for ( int i = 0 ; i < SIZE ; i ++){
        A[ i ] += B [ i ] ;
}
```

Для архитектуры x86 требуется поддержка SSE или более новых версий, SSE - расширение, содержащее 128 битные регистры, в которые можно поместить 4 переменных типа float

В данном примере – нет информационных зависимостей, в результате чего появляется возможность выполнять итерации цикла одновременно

Выравнивание памяти (data alignment)

Для эффективной работы векторных инструкций рекомендуется, чтобы данные в памяти располагались выровненными по определённым адресам, кратным 16, 32 и т.д., в зависимости от разрядности соответствующих векторных инструкций.

Дело в том, что для обработки выровненных упакованных данных, например для SSE, извлечение 16-ти байтовых фрагментов данных будет выполняться за одну операцию.

Если же данные в памяти не выровнены, то процессору потребуется более одной операции для получения необходимого фрагмента данных, что замедлит выполнение.

Например, для SSE есть 2 типа инструкций, для выровненных и невыровненных упакованных данных. Вторые в среднем работают несколько более медленно.

Как выравнивать память?

Для ассемблера есть соответствующие команды (align 16 и т.п.)

В C11 появилась функция aligned_alloc (обычный malloc и realloc не гарантируют выравнивая данных по чему-то большему, чем sizeof (фундаментального типа данных, для которого вызываются)).

В свежей версии POSIX – posix_memalign (_POSIX_VERSION>=200112L), выделенную память можно освобождать с помощью обычной функции free

У Microsoft есть _aligned_malloc, _aligned_free, _aligned_realloc

У компилятора Intel - _mm_malloc()

В GNU C - __attribute__ совместно с aligned (16) // например

int x __attribute__ ((aligned (16))) = 0;

char buffer[256] x __attribute__ ((aligned (16)));

SIMD без OpenMP

- Самый трудоёмкий метод – прямые ассемблерные вставки или ассеблерный код
- Возможно получение хорошей производительности – средствами конкретного компилятора (в частности интринсики (intrinsics) — встроенные объекты компилятора) – например __m128 – обозначение типа данных для SSE в gcc и ряде других компиляторов. Не входят в стандарт C/C++
- Низкоуровневые методы - можно добиться хорошей производительности, но требуется понимание целевой архитектуры, дополнительные трудозатраты для программиста и теряется переностимость
- Использование библиотек, например IPP (Intel Integrated Performance Primitives) – с ограничением этих библиотек
- Прямая поддержка векторизации компилятором

Диалекты языков C/C++

Intrinsics (интринсики SIMD) – расширения языков C/C++ для доступа к векторным инструкциям без ассемблерных вставок.

Для этого, подключаются библиотеки и даются директивы компилятору, что-то вроде:

#pragma GCC target(“avx”)

#pragma GCC optimize(“O3”) // или ключами при компиляции

#include<x86intrin.h>

#include<bits/stdc++.h>

Далее используются соответствующие типы данных, например _m128, для 128 бит или _m256, для 256 бит. По умолчанию целочисленные, если вещественные двойной точности, то d - _m256d и так далее (в соответствии с типом инструкций)

Диалекты языков C/C++

Перед использованием можно также проверить поддержку инструкций CPU, например:

__builtin_cpu_supports(“sse”);

Пусть есть выровненные массивы a, b и c типа double

Далее, как вариант (тут суммируются элементы массивов)

```
for (int i = 0; i < 100; i +=4){
        _m256d x = _mm256_loadu_pd(&a[i]);
        _m256d y = _mm256_loadu_pd(&b[i]);
        _m256d z = _mm256_add_pd(x, y);
        _mm256_storeu_pd(&c[i], z);
}
```

Векторизация - OpenMP SIMD

Для того, чтобы сообщить компилятору, что можно использовать векторные инструкции при распараллеливании того или иного цикла, используется конструкция simd

Формальный синтаксис:

#pragma omp simd [clause[[,] clause] ...],

Может применяться с for (#pragma omp for simd)

Возможные дополнения:

- aligned // aligned([ptr] : [alignment], ...) – сообщаем что указатель выровнен по памяти (дополнительно указывается по какой разрядности)
- reduction // всё как и для обычных циклов

Векторизация - OpenMP SIMD - продолжение

Часто встречающиеся дополнения omp simd, продолжение:

- safelen // safelen([value]), интересная опция, связанная с информационными зависимостями, максимальное безопасное расстояние между итерациями при распараллеливании
- simdlen - желаемое число итераций цикла, подлежащее одновременному выполнению
- collapse //как и для обычных циклов
- linear // linear([variable] : [step], ...) – увеличение с индексом (с шагом)
- private/lastprivate

Пример

```
float A[ SIZE ] , B [ SIZE ] ; //Лучше конечно выровнять по памяти
#pragma omp simd
for ( int i = 0 ; i < SIZE ; i ++){
        A[ i ] += B [ i ] ;
} // выполняется на одном потоке, но с использованием векторных инструкций, возможна и комбинация:
float A[ SIZE ] , B [ SIZE ] ;
#pragma omp parallel for simd
for ( int i = 0 ; i < SIZE ; i ++){
        A[ i ] += B [ i ] ;
} // выполняется параллельно, с использованием ещё и векторных инструкций
```

Пример safelen

```
// по i антизависимость по данным, небезопасно распараллеливать фрагментами больше 4
#pragma omp simd safelen(4)
{
  for (i=0; i<(N-4); i++) {
    a[i] = a[i+4] + b[i] * c[i]; //для a[i]  = a[i-4]+…. тоже будет работать корректно, при пересчёте границ цикла
  }
}
```