ЛАБОРАТОРНАЯ РАБОТА №7
ВЕКТОРИЗАЦИЯ ВЫЧИСЛЕНИЙ

Цели работы

1. Изучение SIMD-расширений архитектуры x86/x86-64.
2. Изучение способов использования SIMD-расширений в программах на языке Си.
3. Получение навыков использования SIMD-расширений.

1. SIMD-РАСШИРЕНИЯ АРХИТЕКТУРЫ x86/x86-64

SIMD-расширения (векторные расширения) были введены во многие стандартные архитектуры с целью повышения скорости обработки потоковых данных. Основная идея SIMD-вычислений заключается в одновременной обработке нескольких элементов данных (вектора) за одну команду.

1.1 Расширение MMX (MultiMedia eXtension)

Первой SIMD-расширение в свой x86-процессор ввела фирма Intel – это было расширение MMX. Оно стало использоваться в процессорах Pentium MMX (расширение архитектуры Pentium или P5) и Pentium II (расширение архитектуры Pentium Pro или P6). Векторное расширение MMX работает с 64-битными регистрами MM0-MM7, логически расположенными на регистрах сопроцессора, и включает 57 новых команд для работы с ними. 64-битные регистры логически могут представляться как одно 64-битное, два 32-битных, четыре 16-битных или восемь 8-битных упакованных целых чисел.

Еще одна особенность технологии MMX – это целочисленная арифметика с насыщением, используемая, например, при обработке графики. В целочисленное арифметике с насыщением переполнение не является

циклическим, как обычно, а вместо этого фиксируется минимальное или максимальное значение. Например, для 8-битного беззнакового целого x:

- обычная арифметика: x=254; x+=3; // результат x=1
- арифметика с насыщением: x=254; x+=3; // результат x=255

1.2 Расширение 3DNow!

Технология 3DNow! была введена фирмой AMD в процессорах K6-2. Это была первая технология, выполняющая потоковую обработку вещественных данных. Расширение работает с 64-битными регистрами MMX, которые теперь представляются как два 32-битных вещественных числа с одинарной точностью. Система команд расширена 21 новой командой, среди которых появилась команда предвыборки данных в кэш L1. В процессорах Athlon и Duron набор команд 3DNow! был несколько дополнен новыми командами для работы с вещественными числами, а также командами MMX и управления кэшированием.

1.3 Расширение SSE (Streaming SIMD Extension)

С процессором Intel Pentium III впервые появилось расширение SSE. Это расширение работает с независимым блоком из восьми 128-битных регистров XMM0-XMM7. Каждый регистр XMM представляет собой четыре упакованных 32-битных вещественных числа с одинарной точностью. Команды блока XMM позволяют выполнять как векторные (над всеми четырьмя значениями регистра), так и скалярные операции (только над одним самым младшим значением). Кроме команд для работы с блоком XMM в расширение SSE входят и дополнительные целочисленные команды для работы с регистрами MMX, а также команды управления кэшированием. В архитектуре x86-64 число регистров XMM было увеличено до 16-ти: XMM0-XMM15.

1.4 Расширение SSE2

В процессоре Intel Pentium4 набор команд получил очередное расширение – SSE2. Это расширение не добавило новых регистров, но позволило по-новому интерпретировать существующие регистры. Расширение SSE2 позволяет работать с 128-битными регистрами XMM как с парой упакованных 64-битных вещественных чисел двойной точности, а также с упакованными целыми числами: 16 байт, 8 слов, 4 двойных (32-битных) слова или 2 учетверенных (64-битных) слова. Соответственно, введены новые команды вещественной арифметики двойной точности и команды целочисленной арифметики: 128-разрядные для регистров XMM и 64-разрядные для регистров MMX. Ряд старых команд MMX распространили и на XMM (в 128-битном варианте). Кроме того, расширена поддержка управления кэшированием и порядком исполнения операций с памятью для многопоточных программ.

1.5 Расширения SSE3, SSSE3, SSE4, ...

В последующих процессорах Intel и AMD происходит дальнейшее расширение системы команд на регистрах MMX и XMM. Добавлены новые команды для ускорения обработки видео, текстовых данных. Особенно следует отметить появившуюся возможность горизонтальной работы с регистрами (выполнение операций с элементами одного вектора).

1.6 Расширение AVX (Advanced Vector Extensions)

В процессорах архитектуры Sandy Bridge от Intel и процессорах архитектуры Bulldozer от AMD векторные расширения сделали следующий большой шаг в развитии: появились новые векторные регистры YMM0-YMM15 размером 256 бит. Существующие ранее регистры XMM стали занимать младшую часть новых регистров. Среди особенностей расширения AVX есть поддержка трехоперандных операций вида (c = a OP b), а также менее строгие требования к выравниванию векторных данных в памяти.

2. ИСПОЛЬЗОВАНИЕ SIMD-РАСШИРЕНИЙ В ПРОГРАММАХ НА ЯЗЫКЕ СИ

Существует несколько способов, позволяющих реализовать возможности имеющихся SIMD-расширений в программах на языках высокого уровня. Условно их можно разделить на ручные, полуавтоматические, автоматические и с помощью готовых библиотек.

2.1. Использование вставок на ассемблере

Многие компиляторы языков Си и Си++ дают возможность вставлять в тело функции команды на ассемблере. Программисту, знакомому с ассемблером, это позволяет контролировать производительность программы в наибольшей степени среди всех описанных в этой лабораторной работе способов. Однако использование вставок затрудняет работу компилятора по оптимизации кода. Кроме этого, теряется переносимость кода, так как: 1) команды во вставке рассчитаны на некоторую конкретную архитектуру, 2) отсутствует стандартный синтаксис ассемблерных вставок, и различные компиляторы используют собственный синтаксис для вставок.

Рассмотрим пример программы для поэлементного сложения двух векторов, состоящих из четырех чисел типа float (см. листинг 1). Программа может быть скомпилирована с помощью компилятора GCC, но не в среде MS Visual Studio, так как там синтаксис ассемблерных вставок отличается.

Листинг 1. Фрагмент программы сложения двух векторов с использованием ассемблерных вставок

```
01  typedef struct{
02          float x, y, z, w;
03  } Vector4;
04  void SSE_Add(Vector4 *res, Vector4 *a, Vector4 *b){
05          asm volatile ("mov %0, %%eax"::"m"(a));
06          asm volatile ("mov %0, %%ebx"::"m"(b));
07          asm volatile ("movups (%eax), %xmm0");
```

```
08          asm volatile ("movups (%ebx), %xmm1");
09          asm volatile ("addps %xmm1, %xmm0");
10          asm volatile ("mov %0, %%eax"::"m"(res));
11          asm volatile ("movups %xmm0, (%eax)");
12  }
```

В строках 05-06 адреса двух векторов загружаются в регистры EAX и EBX. В строках 07-08 оба вектора загружаются в SIMD регистры XMM0 и XMM1 (команда MOVUPS позволяет загружать данные по невыровненным адресам). В строке 09 производится поэлементное сложение двух векторов, а в строке 11 – запись результата в память.

2.2. Использование встроенных SIMD-функций компилятора

Многие современные компиляторы поддерживают встроенные функции (intrinsics). Реализация этих функций встроена в компилятор. Вместо их вызова компилятор вставляет тело функции, т.е. все ее команды. Время исполнения встроенных функций меньше, чем обычных функций, так как операции вызова подпрограммы и возврата из подпрограммы исключены.

Одной из групп встроенных функций являются встроенные функции SIMD-расширений (SIMD intrinsics). Они обеспечивают возможность использования команд SIMD-расширений с помощью привычного синтаксиса для вызова C-функций вместо использования ассемблерного кода и работы с регистрами процессора. В отличие от варианта с использованием ассемблерных вставок, для встроенных функций компилятор выполняет оптимизацию кода.

Рассмотрим пример функции вычисления скалярного произведения двух векторов с помощью SIMD intrinsic (см. листинг 2). Предполагается, что число элементов в векторах кратно четырем.

Листинг 2. Фрагмент программы вычисления скалярного произведения двух векторов с использованием встроенных SIMD-функций компилятора

```
01  #include <xmmintrin.h>
02  float inner2(float* x, float* y, int n){
03    __m128 *xx, *yy;
04    __m128 p, s;
05    xx = (__m128*)x;
06    yy = (__m128*)y;
07    s = _mm_setzero_ps();
08    for(int i=0; i<n/4; ++i){
09      p = _mm_mul_ps(xx[i],yy[i]);
10      s = _mm_add_ps(s,p);
11    }
12    p = _mm_movehl_ps(p,s);
13    s = _mm_add_ps(s,p);
14    p = _mm_shuffle_ps(s,s,1);
15    s = _mm_add_ss(s,p);
16    float sum;
17    _mm_store_ss(&sum,s);
18    return sum;
19  }
```

В строке 01 подключается заголовочный файл с описаниями SIMD intrinsics для расширения SSE. В строке 04 определяются векторные локальные переменные: p для хранения четырех произведений элементов векторов и s для накопления четырех частичных сумм произведений. В строке 07 все четыре частичные суммы устанавливаются в нуль. Каждая итерация цикла в строках 08-11 обрабатывает по четыре элемента исходных векторов. На нулевой итерации обрабатываются элементы 0, 1, 2, 3, на первой – 4, 5, 6, 7 и т.д. В строке 09 вычисляется четыре произведения элементов. В строке 10 эти произведения добавляются к суммам произведений. После выполнения цикла в переменной s хранится четыре частичные суммы (табл. 1). В строках 12-15 эти четыре частичные суммы складываются в одну общую. В строке 17 сумма из векторного регистра записывается в переменную типа float.

Таблица 1

Номер компоненты s	**Частичная сумма, которая хранится в компоненте s**
0	x[0]*y[0] + x[4]*y[4] + x[8]*y[8] +…
1	x[1]*y[1] + x[5]*y[5] + x[9]*y[9] +…
2	x[2]*y[2] + x[6]*y[6] + x[10]*y[10] +…
3	x[3]*y[3] + x[7]*y[7] + x[11]*y[11] +….

2.3 Использование встроенных функций векторных расширений GCC

Компилятор GCC предоставляет средства для описания векторных типов данных и встроенные функции для работы с переменными этих типов. В случае их использования при компиляции исходных текстов программ необходимо указывать ключи, включающие генерацию кода для SSE или SSE2 (`-msse`, `-msse2`). Рассмотрим пример функции, вычисляющей поэлементно квадрат разности значений элементов двух исходных векторов (см. листинг 3).

Листинг 3. Фрагмент программы вычисления квадрат разности значений элементов двух векторов с использованием встроенных функций векторных расширений GCC

```
1  typedef int v4si __attribute__ ((vector_size (16)));
2  v4si sqdif(v4si v1, v4si v2){
3    v4si r;
4    r = a - b;
5    r *= r;
6    return r;
7  }
```

В строке 1 с помощью атрибута определяется векторный тип данных. Переменные этого типа содержат четыре элемента, являющиеся целыми знаковыми числами (типа `int`). Размер вектора – 16 байт – указан в параметре атрибута. В строке 2 указан заголовок функции, принимающей два

векторных параметра и возвращающей вектор. В строке 3 объявлена локальная векторная переменная. В строке 4 поэлементно вычисляется разность значений в двух исходных векторах, а результат записывается в локальную переменную. В строке 5 все элементы вектора возводятся в квадрат. В строке 6 в качестве результата функции возвращается получившийся вектор.

Главный недостаток данного способа векторизации – привязка к компилятору GCC. К достоинствам относится отсутствие привязки к конкретной архитектуре. То есть, возможно написание текста, который будет без модификации исполняться на различных архитектурах, имеющих SIMD-расширения. Кроме этого, у текста программы, написанного с использованием данного способа, читаемость выше, чем, например, в варианте с ассемблерными вставками и SIMD intrinsics.

Подытожим сравнение трех способов векторизации. Использование ассемблерных вставок позволяет вручную строить наиболее эффективные программы. Вариант с использованием SIMD intrinsics позволяет достигать почти такой же эффективности полуавтоматически средствами компилятора, без необходимости использования машинных команд и работы с регистрами. Оба эти варианта привязаны к одной архитектуре процессора: x86/x86-64. Этот недостаток преодолевается с помощью встроенных функций векторных расширений GCC. Данный способ подходит для разных архитектур, имеющих SIMD-команды. Кроме этого, программы, построенные с его использованием, обладают наилучшей читаемостью среди всех рассмотренных способов. Существенный недостаток этого способа заключается в привязке к компилятору GCC.

2.4 Использование автоматической векторизации компилятором

Оптимизирующий компилятор, умеющий из обычного кода генерировать код для SIMD-расширения – это наиболее простой и эффективный путь к достижению высокой производительности. Основным

недостатком этого подхода является то, что код должен удовлетворять определенным критериям, чтобы компилятор мог его векторизовать. Во многих случаях компилятор не может распознать возможность эффективного применения векторных операций.

Некоторые компиляторы (например, Intel C/C++ Compiler) поддерживают специальные директивы, с помощью которых программист может дать компилятору дополнительную информацию о коде, способствующую его векторизации. Например, директива может указывать компилятору, что итерации некоторого цикла независимы друг от друга, и их можно выполнять параллельно.

2.5 Использование высокопроизводительных библиотек

Для многих предметных областей существуют эффективно реализованные библиотеки операций, оптимизированные под различные вычислительные системы. Наиболее ярким примером таких библиотек можно назвать BLAS и LAPACK, которые содержат процедуры, реализующие многие операции линейной алгебры (работа с векторами, матрицами).

Существуют реализации этих библиотеки под многие архитектуры, что обеспечивает переносимость программ, написанных с их использованием. Например, реализацию BLAS содержат библиотеки Intel MKL, AMD ACML, NVIDIA CuBLAS, свободно распространяемая библиотека ATLAS. Каждая реализация стремится учесть все особенности целевой архитектуры для достижения высокой производительности. В частности, для временного хранения данных эффективно используется кэш-память, а для вычислений используются векторные операции.

3. ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Написать три варианта программы, реализующей алгоритм из задания:
   - вариант без ручной векторизации,
   - вариант с ручной векторизацией (выбрать любой вариант из возможных трех: ассемблерная вставка, встроенные функции компилятора, расширение GCC),
   - вариант с матричными операциями, выполненными с использованием оптимизированной библиотеки BLAS.

   Для элементов матриц использовать тип данных float.
2. Проверить правильность работы программ на нескольких небольших тестовых наборах входных данных.
3. Каждый вариант программы оптимизировать по скорости, насколько это возможно.
4. Сравнить время работы трех вариантов программы для N=2048, M=10.
5. Составить отчет по лабораторной работе. Отчет должен содержать следующее:
   - Титульный лист.
   - Цель лабораторной работы.
   - Результаты измерения времени работы трех программ.
   - Полный компилируемый листинг реализованных программ и команды для их компиляции.
   - Вывод по результатам лабораторной работы.

4. ВАРИАНТЫ ЗАДАНИЙ

1. Алгоритм обращения матрицы A размером $N \times N$ с помощью разложения в ряд: $A^{-1} = (I + R + R^2 + \ldots)B$, где $R = I - BA$, $B = \frac{A^T}{\|A\|_1 \cdot \|A\|_\infty}$, $\|A\|_1 = \max_j \sum_i |A_{ij}|$, $\|A\|_\infty = \max_i \sum_j |A_{ij}|$, I – единичная матрица (на главной диагонали – единицы, остальные – нули). Параметры алгоритма: N – размер матрицы, M – число членов ряда (число итераций цикла в реализации алгоритма).

5. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое SIMD-расширение архитектуры? Зачем вводятся SIMD-расширения обычных архитектур?
2. Какие SIMD-расширения архитектуры x86/x86-64 сейчас существуют? С какими регистрами и типами данных работает каждое SIMD-расширение?
3. Какие существуют способы использования SIMD-расширений в программах на языке Си?