МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНОЙ РАБОТЫ №5

Лабораторная работа №5 посвящена изучению SIMD-расширений архитектуры x86.

Типы параллелизма

Параллелизм на уровне битов.

Эта форма параллелизма основана на **увеличении размера машинного слова**. Увеличение размера машинного слова уменьшает количество операций, необходимых процессору для выполнения действий над переменными, чей размер превышает размер машинного слова. К примеру: на 8-битном процессоре нужно сложить два 16-битных целых числа. Для этого вначале нужно сложить младшие 8 бит чисел, затем сложить старшие 8 бит и к результату их сложения прибавить значение флага переноса. Итого 3 инструкции. С 16-битным процессором можно выполнить эту операцию одной инструкцией.

Параллелизм на уровне инструкций.

Компьютерную программу можно рассматривать как последовательный поток инструкций, выполняемых процессором. Однако порядок этих инструкций можно изменить, распределить их по группам, которые будут выполняться параллельно, без изменения результата работы всей программы. Такой тип параллелизма называется **параллелизмом на уровне инструкций**.

Для реализации данного вида параллелизма в микропроцессорах используется несколько **конвейеров команд**, технологии **предсказания команд** и **переименования регистров**.

Параллелизм на уровне данных.

Основная идея подхода, основанного на параллелизме данных, заключается в том, что **одна операция выполняется сразу над всеми элементами массива данных**. Различные фрагменты такого массива обрабатываются на векторном процессоре или на разных процессорах параллельной машины. Распределением данных между процессорами занимается программа. Векторизация или распараллеливание в этом случае чаще всего выполняется уже на этапе компиляции – перевода исходного текста программы в машинные команды.

Данный вид параллелизма широко используется при решении задач численного моделирования.

Параллелизм на уровне задач.

Когда одна большая задача может быть разбита на ряд независимых подзадач, каждую из которых можно решить отдельно, допустимо распараллеливание на уровне задач.

SIMD-расширения

SIMD-расширения (Single Instruction Multiple Data) были введены в архитектуру x86 с целью повышения скорости обработки потоковых данных.

Основная идея заключается в одновременной обработке нескольких элементов данных за одну инструкцию (то есть распараллеливание на уровне данных).

Расширение MMX.

Первой SIMD-расширение в свой x86-процессор ввела фирма Intel – это расширение MMX. Оно стало использоваться в процессорах Pentium MMX и Pentium II. Расширение MMX работает с 64-битными регистрами MM0-MM7, физически расположенными на регистрах сопроцессора, и включает 57 новых инструкций 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. Еще одна особенность технологии MMX – это арифметика с насыщением. При этом переполнение не является циклическим, как обычно, а 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
```

Расширение 3DNow!

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

Расширение SSE.

С процессором Intel Pentium III впервые появилось расширение SSE (Streaming SIMD Extension). 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 инструкций с блоком XMM в расширение SSE входят и дополнительные целочисленные инструкции с регистрами MMX, а также инструкции управления кэшированием.

Расширение SSE2

В процессоре Intel Pentium 4 набор инструкций получил очередное расширение - SSE2. Оно 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 слова или 2 учетверенных (64-битных) слова. Введены новые инструкции вещественной арифметики двойной точности, инструкции целочисленной арифметики, 128-разрядные для регистров XMM и 64-разрядные для регистров MMX. Ряд старых инструкций MMX распространили и на XMM (в 128-битном варианте). Кроме того, расширена поддержка управления кэшированием и порядком исполнения операций с памятью.

Расширение SSE3

Третья версия SIMD-расширения Intel, потомок SSE, SSE2 и MMX. Впервые представлено 2 февраля 2004 года в ядре Prescott процессора Pentium 4. В 2005 AMD предложила свою реализацию SSE3 для процессоров Athlon 64 (ядра Venice, San Diego и Newark).

SSE в языке C

Объявления встроенных функций (intrinsics) SSE содержатся в заголовочном файле xmmintrin.h.

Для работы с векторными данными, содержащими несколько упакованных значений, в языках C/C++ используются следующие типы данных:

__m64 – 64-бит (регистр MMX):

- 1 * 64-битное целое;
- 2 * 32-битных целых;
- 4 * 16-битных целых;
- 8 * 8-битных целых.

__m128 – 128-бит (регистр XMM):

- 4 * 32-битных вещественных (SSE);
- 2 * 64-битных вещественных (SSE2).

Элементы таких типов данных (для наибольшей эффективности) должны быть выровнены в памяти по соответствующей границе. Например, начало массива элементов типа __m64 выравнивается по 8 байтам, а массив элементов __m128 – по 16 байтам. Статические переменные и массивы компилятор выравнивает автоматически. Динамические данные компилятор обычно выравнивает по только величине 4 байта. Если данные векторных типов оказались невыровненными, то для работы с ними следует применять специальные команды невыровненного чтения и записи (они работают медленнее выровненных). Для выделения памяти с выравниванием используется функция:

void *_mm_malloc(int size, int align)
size - объем выделяемой памяти в байтах (как в malloc),
align - выравнивание в байтах.

Для освобождения памяти, выделенной таким образом, используется функция:

void _mm_free(void *p);

Арифметические операции:

Функция	Инструкция	Операция	R0	R1	R2	R3
_mm_add_ss	ADDSS	сложение	a0 [op] b0	a1	a2	a3
_mm_add_ps	ADDPS	сложение	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
_mm_sub_ss	SUBSS	вычитание	a0 [op] b0	a1	a2	a3
_mm_sub_ps	SUBPS	вычитание	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
_mm_mul_ss	MULSS	умножение	a0 [op] b0	a1	a2	a3
_mm_mul_ps	MULPS	умножение	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
_mm_div_ss	DIVSS	деление	a0 [op] b0	a1	a2	a3
_mm_div_ps	DIVPS	деление	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
_mm_sqrt_ss	SQRTSS	квадратный корень	[op] a0	a1	a2	a3
_mm_sqrt_ps	SQRTPS	квадратный корень	[op] a0	[op] b1	[op] b2	[op] b3
_mm_rcp_ss	RCPSS	обратное значение	[op] a0	a1	a2	a3
_mm_rcp_ps	RCPPS	обратное значение	[op] a0	[op] b1	[op] b2	[op] b3
_mm_rsqrt_ss	RSQRTSS	обратное значение квадратного корня	[op] a0	a1	a2	a3
_mm_rsqrt_ps	RSQRTPS	обратное значение квадратного корня	[op] a0	[op] b1	[op] b2	[op] b3
_mm_min_ss	MINSS	минимум	[op](a0,b0)	a1	a2	a3
_mm_min_ps	MINPS	минимум	[op](a0,b0)	[op](a1,b1)	[op](a2,b2)	[op](a3,b3)
_mm_max_ss	MAXSS	максимум	[op](a0,b0)	a1	a2	a3
_mm_max_ps	MAXPS	максимум	[op](a0,b0)	[op](a1,b1)	[op](a2,b2)	[op](a3,b3)

Логические операции:

Функция	Инструкция	Операция
_mm_and_ps	ANDPS	побитовое И
_mm_andnot_ps	ANDNPS	побитовое И-НЕ
_mm_or_ps	ORPS	побитовое ИЛИ
_mm_xor_ps	XORPS	побитовое исключающее ИЛИ

Команды для инициализации и работы с памятью

Функция	Инструкция	Операция
_mm_load_ss	MOVSS	загрузить младшее значение и очистить остальные три значения
_mm_load1_ps	MOVSS + Shuffling	загрузить одно значение во все четыре позиции
_mm_load_ps	MOVAPS	Загрузить четыре значения по выровненному адресу
_mm_loadu_ps	MOVUPS	Загрузить четыре значения по невыровненному адресу
_mm_loadr_ps	MOVAPS + Shuffling	Загрузить четыре значения в обратном порядке
_mm_set_ss	составная	устанавливает самое младшее значение и обнуляет три остальных
_mm_set1_ps	составная	устанавливает четыре позиции в одно значение
_mm_set_ps	составная	устанавливает четыре значения, выровненные по адресу
_mm_setr_ps	составная	устанавливает четыре значения в обратном порядке
_mm_setzero_ps	составная	Обнуляет все четыре значения
_mm_store_ss	MOVSS	записать младшее значение
_mm_store1_ps	MOVSS + Shuffling	записать младшее значение во все четыре позиции
_mm_store_ps	MOVAPS	записать четыре значения по выровненному адресу
_mm_storeu_ps	MOVUPS	записать четыре значения по невыровненному адресу
_mm_storer_ps	MOVAPS + Shuffling	записать четыре значения в обратном порядке
_mm_move_ss	MOVSS	записать младшее значение и оставить без изменения три остальных значения

Контрольные вопросы

1. Какие типы параллелизма вы знаете? В чем идея каждого из них?
2. Что такое SIMD-расширения? Какие SIMD-расширения вы знаете?
3. Какие типы данных используются при работе с расширениями SSE на языках C/C++? Что необходимо учесть при их использовании?

ПРИЛОЖЕНИЕ

Листинг программы для выполнения домашней подготовки ЛР5.

```
#include <stdio.h>
#include <xmmintrin.h>
#include <time.h>

#define N 400000

// «обычная» функция
float inner1(float *x, float *y, int n)
{
  float s = 0;
  int i;
  for(i=0; i<n; i++)
    s += x[i]*y[i];
  return s;
}

// функция с использованием SSE
float inner2(float *x, float *y, int n)
{
  float sum;
  int i;
  __m128 *xx, *yy;
    m128 p, s;
  xx = ( __m128 *)x;
  yy = ( __m128 *)y;
  s = _mm_set_ps1(0);
  for (i=0; i<n/4; i++)
  {
    p = _mm_mul_ps(xx[i], yy[i]);    // векторное умножение четырех чисел
    s = _mm_add_ps(s, p);            // векторное сложение четырех чисел
  }
  p = _mm_movehl_ps(p, s);              // перемещение двух старших значений s в младшие p
  s = _mm_add_ps(s, p);                 // векторное сложение
  p = _mm_shuffle_ps(s, s, 1);          // перемещение второго значения в s в младшую позицию в p
  s = _mm_add_ss(s, p);                 // скалярное сложение
  _mm_store_ss(&sum, s);                // запись младшего значения в память;
  return sum;
}

int main()
{
  float *x, *y, s;
  long t;
  int i;
  x = (float *)_mm_malloc(N*sizeof(float), 16);  // выделение памяти с выравниванием
  y = (float *)_mm_malloc(N*sizeof(float), 16);  // выделение памяти с выравниванием

  for (i=0; i<N; i++)
  {
    x[i] = 10;            // заполнение массивов
    y[i] = 10;            // заполнение массивов
  }
  // Using x87
  double time_spent = 0.0;
  clock_t begin = clock();
  for(int itt1=0; itt1<1000; itt1++)
    s = inner1(x, y, N);
  printf("Result: %f\n", s);
  clock_t end = clock();
  time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
  printf("The elapsed time is %f seconds\n", time spent);
  // Using SSE
  time_spent = 0.0;
  begin = clock();
  for(int itt2=0; itt2<1000; itt2++)
    s = inner2(x, y, N);
  printf("Result: %f\n", s);
  end = clock();
  time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
  printf("The elapsed time is %f seconds\n", time_spent);
  _mm_free(x);
  _mm_free(y);

  return 0;
}
```