РЕАЛИЗАЦИЯ И ОЦЕНКА ПРОИЗВОДИТЕЛЬНОСТИ ОБРАБОТКИ ДАННЫХ ПОСРЕДСТВОМ ВЕКТОРНЫХ ВЫЧИСЛЕНИЙ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ C

Азаренко Р. В. (polytech@basicxp.ru)
Московский политехнический университет, Факультет информатики и систем управления
Научный руководитель: ст. преп. Акимов В. А.
УДК: 004.42+519.683

В данной статье рассматривается отличие векторных вычислений от скалярных, реализация векторных вычислений для центральных процессоров архитектур IA-32 и x86-64 на языке программирования C с использованием intrinsics на примере сложения двух числовых массивов и сравнение производительности скалярных и векторных вычислений.

Ключевые слова: векторные вычисления, SIMD, обработка данных, программирование.

IMPLEMENTATION AND PERFORMANCE EVALUATION OF DATA PROCESSING USING VECTOR COMPUTATIONS IN C PROGRAMMING LANGUAGE

Roman Azarenko (polytech@basicxp.ru)
Moscow Polytechnic University, Faculty of Informatics and Control systems
Scientific advisor: sr. lecturer Valentin Akimov

This article discusses the difference between vector and scalar computations, provides an implementation of a vector computation algorithm for central processing units with IA-32 and x86-64 architectures in C programming language for adding two numeric arrays using intrinsics, and compares performance of scalar and vector computations.

Keywords: vector computing, SIMD, data processing, programming.

Постановка задачи

Векторные вычисления — это вид параллельных вычислений с параллелизмом на уровне данных (SIMD). Цель векторизации кода: ускорить работу программы и уменьшить объём кода. Одна векторная команда распознаётся, декодируется и выполняется быстрее нескольких скалярных, выполняющих то же действие, а также занимает меньше места в программе и в различных очередях/таблицах/буферах в процессоре. [1]

Рассмотрим практическую реализацию векторной обработки данных на примере следующей задачи. Пусть имеется два вектора (одномерных массива) A и B размером n = 179 целочисленных элементов каждый:

```
A = [32767, 122, 284, 19, 323, 322, 129, 764, 524, 195, 581, 641, 669, 788, 268,
338, 979, 494, 403, 721, 854, 376, 486, 163, 508, 92, 17, 770, 29, 246, 774,
761, 587, 197, 652, 192, 564, 763, 623, 378, 765, 325, 890, 480, 545, 260, 625,
663, 835, 525, 860, 918, 69, 240, 319, 239, 612, 936, 977, 367, 63, 229, 588,
772, 795, 53, 999, 834, 627, 660, 210, 237, 634, 358, 941, 954, 958, 218, 579,
639, 570, 680, 872, 938, 758, 708, 878, 45, 791, 7, 713, 110, 969, 937, 409,
593, 133, 550, 728, 393, 52, 781, 929, 282, 224, 869, 924, 813, 425, 473, 946,
988, 730, 900, 114, 989, 41, 964, 842, 947, 331, 675, 704, 373, 912, 630, 615,
343, 38, 187, 965, 402, 98, 149, 411, 801, 831, 598, 242, 666, 33, 633, 933,
820, 804, 109, 717, 877, 59, 735, 266, 561, 270, 827, 304, 175, 573, 719, 188,
91, 206, 773, 673, 674, 340, 235, 264, 391, 327, 982, 906, 21, 746, 909, 546,
385, 405, 497, 345]

B = [2, 449, 676, 917, 602, 933, 165, 370, 492, 644, 624, 46, 8, 125, 159, 59,
451, 101, 577, 737, 382, 931, 462, 71, 603, 204, 661, 85, 241, 585, 962, 866,
761, 826, 748, 108, 146, 923, 768, 539, 836, 364, 144, 738, 926, 132, 912, 892,
149, 83, 660, 754, 39, 674, 4, 663, 211, 444, 152, 776, 847, 831, 869, 570, 797,
404, 942, 43, 471, 793, 733, 641, 169, 905, 822, 622, 788, 69, 189, 193, 610,
```

```
627, 669, 281, 535, 789, 650, 813, 853, 531, 304, 818, 72, 564, 250, 300, 877,
129, 595, 28, 759, 638, 378, 885, 425, 198, 675, 500, 731, 634, 7, 790, 226,
799, 551, 221, 158, 44, 483, 216, 473, 666, 693, 202, 344, 868, 777, 57, 229,
330, 461, 967, 889, 424, 63, 282, 631, 863, 254, 316, 386, 173, 78, 941, 283,
58, 997, 133, 683, 143, 934, 646, 350, 710, 798, 9, 195, 943, 555, 209, 755,
548, 672, 591, 470, 140, 872, 814, 42, 48, 495, 916, 516, 265, 487, 758, 615,
309, 546]
```

Необходимо получить вектор (одномерный массив) C размера n, содержащий поэлементные суммы векторов A и B. Необходимо произвести вычисления максимально эффективным способом, учитывая особенности программного и аппаратного обеспечения.

Стратегия решения задачи

Осуществим выполнение данной задачи как поэлементно (скалярно), так и при помощи векторной обработки.

При скалярном решении выполнение вычислений происходит в один поток, то есть в каждый момент времени обрабатывается только одна пара чисел. Так как имеем n пар элементов, то время выполнения будет прямо пропорционально величине n.

При векторном решении выполнение вычислений происходит сразу над несколькими парами элементов, при этом количество этих пар зависит от способа представления чисел в памяти, а также используемой технологии векторных вычислений.

В рамках данной работы для простоты примем размер чисел в каждой из пар равным 16 бит (2 байта). Также для составления сравнительной характеристики примем количество элементов n кратным размеру используемых регистров. Случай некратного количества элементов будет обработан в коде путём обработки невыровненного остатка скалярным способом.

Атомарный размер элементов векторной обработки и сложность вычислений в O-нотации при заданных алгоритмах приведена в таблице 1.

Таблица 1. Сравнительная характеристика технологий векторных вычислений

Технология	Атомарный размер, тип регистра	Сложность вычислений
— (скалярно)	по архитектуре процессора	$O(n)$
MMX	64 бит (`mm`, только архитектура IA-32)	$O(n/4)$
SSE2	128 бит (`xmm`)	$O(n/8)$
AVX2	256 бит (`ymm`)	$O(n/16)$
AVX-512	512 бит (`zmm`)	$O(n/32)$

Рассмотрим решение данной задачи как скалярно, так и при помощи технологий SSE2 и AVX2. [2]. Технология MMX на сегодняшний день считается устаревшей, поэтому она не была рассмотрена, а аппаратная реализация AVX-512 на момент выполнения данной практической работы недоступна для домашнего использования [3].

Блок-схемы алгоритмов

Составим блок-схемы алгоритмов для выбранных методов выполнения вычислений. На рисунке 1 слева показан скалярный метод, по центру — SSE2, справа — AVX2.

Скалярный метод решения задачи наверняка известен всем из различных учебников по программированию, где читатель обучается работе с массивами. Данный способ самый простой в исполнении и понимании, но одновременно и самый неэффективный.

В любом из векторных методов мы используем указатели на текущее положение в массиве и сдвигаем их на атомарный размер регистров, соответствующих используемой технологии. Заметьте, что в случае с SSE2 и AVX2 отличаются только используемые регистры и вызываемые инструкции, а сам принцип действия остаётся одним и тем же.

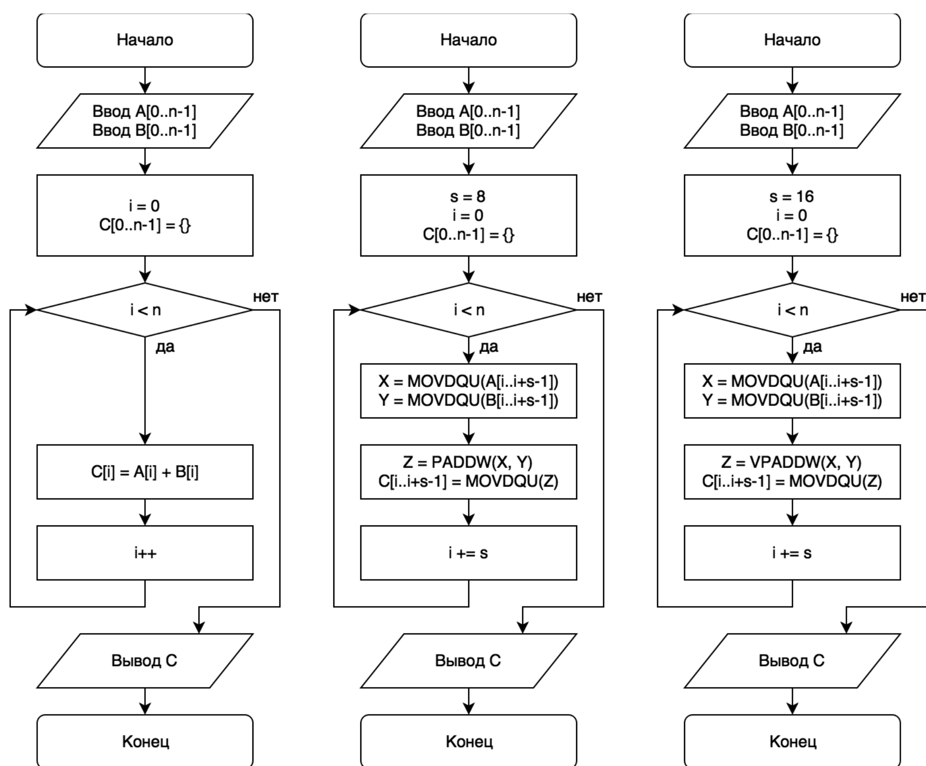

Рисунок 1. Блок-схемы выбранных методов решения

Составление программы

Создадим программу на языке C с использованием встроенных (платформенных) функций (intrinsics) компилятора Clang [4] для решения задачи. Данный код также был протестирован с компилятором языка C GCC [5].

Для краткости приведём значимые фрагменты программы, отвечающие за сами вычисления. Полных исходный код демонстрационной программы приведён в разделе «Исходный код программы» ниже.

На вход функций подаются указатели на два входных одномерных массива (вектора) `*a` и `*b` и на выходной одномерный массив (вектор) `*c`, а также их размер `len`.
Следующая функция решает задачу скалярно:

```
void add_loop(t_int* a, t_int* b, t_int* c, t_int len) {
    for (t_int i = 0; i < len; ++i)
        c[i] = a[i] + b[i];
}
```

Приведённая ниже функция решает задачу векторно с использованием технологии SSE2:

```
void add_sse2(t_int* a, t_int* b, t_int* c, t_int len) {
    const int block_size = 16 / sizeof(t_int);
    const int len_aligned = len - len % block_size;
```

```
    for (int i = 0; i < len_aligned; i += block_size) {
        _mm_storeu_si128(
            (__m128i*) &c[i],
            _mm_add_epi16(
                _mm_loadu_si128((__m128i*) &a[i]),
                _mm_loadu_si128((__m128i*) &b[i])
            )
        );
    }

    for (int i = len_aligned; i < len; ++i)
        c[i] = a[i] + b[i];
}
```

Наконец, функция ниже решает задачу векторно с использованием технологии AVX2:

```
void add_avx2(t_int* a, t_int* b, t_int* c, t_int len) {
    const int block_size = 32 / sizeof(t_int);
    const int len_aligned = len - len % block_size;

    for (int i = 0; i < len_aligned; i += block_size)
        _mm256_storeu_si256(
            (__m256i*) &c[i],
            _mm256_add_epi16(
                _mm256_loadu_si256((__m256i*) &a[i]),
                _mm256_loadu_si256((__m256i*) &b[i])
            )
        );

    for (int i = len_aligned; i < len; ++i)
        c[i] = a[i] + b[i];
}
```

Тестирование производительности

После написания программы, она была протестирована для оценки скорости и качества её работы. Все три реализации вычислений возвращают корректный результат. При выполнении большого количества итераций заметна разница в производительности использованных способов решения.

Выполним 10 000 000 итераций данной программы на центральном процессоре общего назначения Intel Xeon E5-2650Lv3 [6] во всех поддерживаемых режимах и сравним их производительности.

Вывод программы при выполнении вычислений с использованием набора инструкций SSE2 выглядит как показано ниже:

```
Вычисление при помощи SSE2 (векторно)
32769, 571, 960, 936, 925, 1255, 294, 1134, 1016, 839, 1205, 687, 677, 913, 427,
397, 1430, 595, 980, 1458, 1236, 1307, 948, 234, 1111, 296, 678, 855, 270, 831,
1736, 1627, 1348, 1023, 1400, 300, 710, 1686, 1391, 917, 1601, 689, 1034, 1218,
1471, 392, 1537, 1555, 984, 608, 1520, 1672, 108, 914, 323, 902, 823, 1380,
1129, 1143, 910, 1060, 1457, 1342, 1592, 457, 1941, 877, 1098, 1453, 943, 878,
803, 1263, 1763, 1576, 1746, 287, 768, 832, 1180, 1307, 1541, 1219, 1293, 1497,
1528, 858, 1644, 538, 1017, 928, 1041, 1501, 659, 893, 1010, 679, 1323, 421,
811, 1419, 1307, 1167, 649, 1067, 1599, 1313, 1156, 1107, 953, 1778, 956, 1699,
665, 1210, 199, 1008, 1325, 1163, 804, 1341, 1397, 575, 1256, 1498, 1392, 400,
267, 517, 1426, 1369, 987, 573, 474, 1083, 1462, 1461, 496, 982, 419, 806, 1011,
1761, 1087, 167, 1714, 1010, 742, 878, 1200, 1207, 620, 1537, 1102, 184, 768,
1662, 743, 300, 961, 1321, 1345, 1265, 810, 375, 1136, 1205, 369, 1030, 1401,
937, 1262, 1174, 1033, 1143, 1020, 806, 891
```

Время выполнения было измерено при помощи стандартной утилиты time [7] в Linux. Каждый тест был выполнен 3 раза и было в качестве результата было взято среднее арифметическое. Результаты измерений приведены в таблице 2.

Таблица 2. Результаты тестирования производительности

Способ решения	**Время выполнения, сек.**
Скалярный	8,472
SSE2 (векторный)	1,970
AVX2 (векторный)	1,129

Заключение

Использование векторных инструкций при выполнении большого количества однообразных вычислений позволяет существенно сократить время обработки данных. Почти с каждым новым поколением центральных процессоров добавляются новые инструкции и регистры, позволяющие более эффективно обрабатывать большие массивы данных.

Исходный код программы

```
// Количество итераций (для тестирования времени выполнения)
#define ITER_COUNT 10000000

// Количество элементов массивов и их содержание
#define ELEM_COUNT 179
#define ELEM_A 32767, 122, 284, 19, 323, 322, 129, 764, 524, 195, 581, 641, 669,
788, 268, 338, 979, 494, 403, 721, 854, 376, 486, 163, 508, 92, 17, 770, 29,
246, 774, 761, 587, 197, 652, 192, 564, 763, 623, 378, 765, 325, 890, 480, 545,
260, 625, 663, 835, 525, 860, 918, 69, 240, 319, 239, 612, 936, 977, 367, 63,
229, 588, 772, 795, 53, 999, 834, 627, 660, 210, 237, 634, 358, 941, 954, 958,
218, 579, 639, 570, 680, 872, 938, 758, 708, 878, 45, 791, 7, 713, 110, 969,
937, 409, 593, 133, 550, 728, 393, 52, 781, 929, 282, 224, 869, 924, 813, 425,
473, 946, 988, 730, 900, 114, 989, 41, 964, 842, 947, 331, 675, 704, 373, 912,
630, 615, 343, 38, 187, 965, 402, 98, 149, 411, 801, 831, 598, 242, 666, 33,
633, 933, 820, 804, 109, 717, 877, 59, 735, 266, 561, 270, 827, 304, 175, 573,
719, 188, 91, 206, 773, 673, 674, 340, 235, 264, 391, 327, 982, 906, 21, 746,
909, 546, 385, 405, 497, 345
#define ELEM_B 2, 449, 676, 917, 602, 933, 165, 370, 492, 644, 624, 46, 8, 125,
159, 59, 451, 101, 577, 737, 382, 931, 462, 71, 603, 204, 661, 85, 241, 585,
962, 866, 761, 826, 748, 108, 146, 923, 768, 539, 836, 364, 144, 738, 926, 132,
912, 892, 149, 83, 660, 754, 39, 674, 4, 663, 211, 444, 152, 776, 847, 831, 869,
570, 797, 404, 942, 43, 471, 793, 733, 641, 169, 905, 822, 622, 788, 69, 189,
193, 610, 627, 669, 281, 535, 789, 650, 813, 853, 531, 304, 818, 72, 564, 250,
300, 877, 129, 595, 28, 759, 638, 378, 885, 425, 198, 675, 500, 731, 634, 7,
790, 226, 799, 551, 221, 158, 44, 483, 216, 473, 666, 693, 202, 344, 868, 777,
57, 229, 330, 461, 967, 889, 424, 63, 282, 631, 863, 254, 316, 386, 173, 78,
941, 283, 58, 997, 133, 683, 143, 934, 646, 350, 710, 798, 9, 195, 943, 555,
209, 755, 548, 672, 591, 470, 140, 872, 814, 42, 48, 495, 916, 516, 265, 487,
758, 615, 309, 546

#include <immintrin.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>

typedef uint16_t t_int;

// Вывод содерживого массива типа t_int
void printarr (t_int* arr, t_int len) {
    for (t_int i = 0; i < len; ++i) {
        printf("%d", arr[i]);
```

```
        if (i < len - 1)
            printf(", ");
    }

    printf("\n");
}

// Скалярное сложение двух массивов с помощью цикла
void add_loop(t_int* a, t_int* b, t_int* c, t_int len) {
    for (t_int i = 0; i < len; ++i)
        c[i] = a[i] + b[i];
}

// Векторное сложение двух массивов с помощью SSE2
void add_sse2(t_int* a, t_int* b, t_int* c, t_int len) {
    const int block_size = 16 / sizeof(t_int);
    const int len_aligned = len - len % block_size;

    for (int i = 0; i < len_aligned; i += block_size) {
        _mm_storeu_si128(
            (__m128i*) &c[i],
            _mm_add_epi16(
                _mm_loadu_si128((__m128i*) &a[i]),
                _mm_loadu_si128((__m128i*) &b[i])
            )
        );
    }

    for (int i = len_aligned; i < len; ++i)
        c[i] = a[i] + b[i];
}

// Векторное сложение двух массивов с помощью AVX2
void add_avx2(t_int* a, t_int* b, t_int* c, t_int len) {
    const int block_size = 32 / sizeof(t_int);
    const int len_aligned = len - len % block_size;

    for (int i = 0; i < len_aligned; i += block_size)
        _mm256_storeu_si256(
            (__m256i*) &c[i],
            _mm256_add_epi16(
                _mm256_loadu_si256((__m256i*) &a[i]),
                _mm256_loadu_si256((__m256i*) &b[i])
            )
        );

    for (int i = len_aligned; i < len; ++i)
        c[i] = a[i] + b[i];
}

int main(int argc, char* argv[]) {
    t_int a[ELEM_COUNT] = {ELEM_A};
    t_int b[ELEM_COUNT] = {ELEM_B};
    t_int c[ELEM_COUNT];

    char cmds[] = "loop, sse2, avx2";

    if (argc != 2) {
        printf("Укажите способ сложения массивов. Возможные варианты: %s.\n",
cmds);
        return 1;
    }

    if (strcmp(argv[1], "loop") == 0) {
```

```
        printf("Вычисление при помощи цикла (скалярно)\n");

        for (uint32_t i = 0; i < ITER_COUNT; ++i)
            add_loop(a, b, c, ELEM_COUNT);
    }
    else if (strcmp(argv[1], "sse2") == 0) {
        printf("Вычисление при помощи SSE2 (векторно)\n");

        for (uint32_t i = 0; i < ITER_COUNT; ++i)
            add_sse2(a, b, c, ELEM_COUNT);
    }
    else if (strcmp(argv[1], "avx2") == 0) {
        printf("Вычисление при помощи AVX2 (векторно)\n");

        for (uint32_t i = 0; i < ITER_COUNT; ++i)
            add_avx2(a, b, c, ELEM_COUNT);
    }
    else {
        printf("Указан неверный способ сложения массивов. Возможные варианты:
%s.\n", cmds);
        return 2;
    }

    printarr(c, ELEM_COUNT);

    return 0;
}
```

Список использованных источников

1. Киреев С. Е., Калгин К. В. «Эффективное программирование современных микропроцессоров и мультипроцессоров. Векторизация вычислениц» — Новосибирск: 2015, НГУ
2. Intel® 64 and IA-32 Architectures Software Developer's Manual — Санта-Клара: 2016, Intel Corporation — 4670 с. — http://www.intel.com/content/www/us/en/processors/architectures- software-developer-manuals.html
3. Википедия: свободная энциклопедия. AVX-512 — https://en.wikipedia.org/wiki/AVX-512#CPUs_with_AVX-512
4. Документация по компилятору LLVM Clang — http://clang.llvm.org/doxygen/index.html
5. GNU Compiler Collection — https://gcc.gnu.org
6. Intel ARK. Intel Xeon E5-2650Lv3 — http://ark.intel.com/products/81903
7. GNU project. time — https://www.gnu.org/software/time/