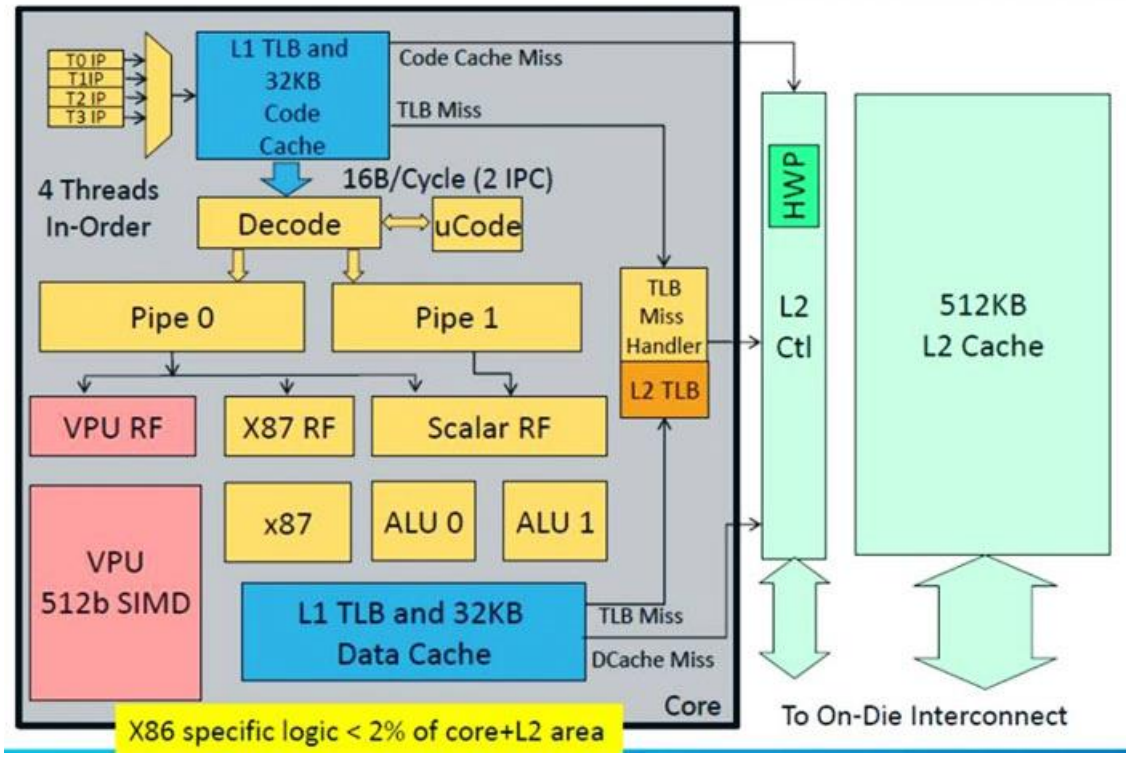
Векторизация кода для Intel® Xeon Phi™ с помощью функций-интринсиков

Виктор Гетманский
Ефим Сергеев
Олег Шаповалов
Дмитрий Крыжановский

Singularis Lab, Ltd.
Volgograd State Technical University
2015

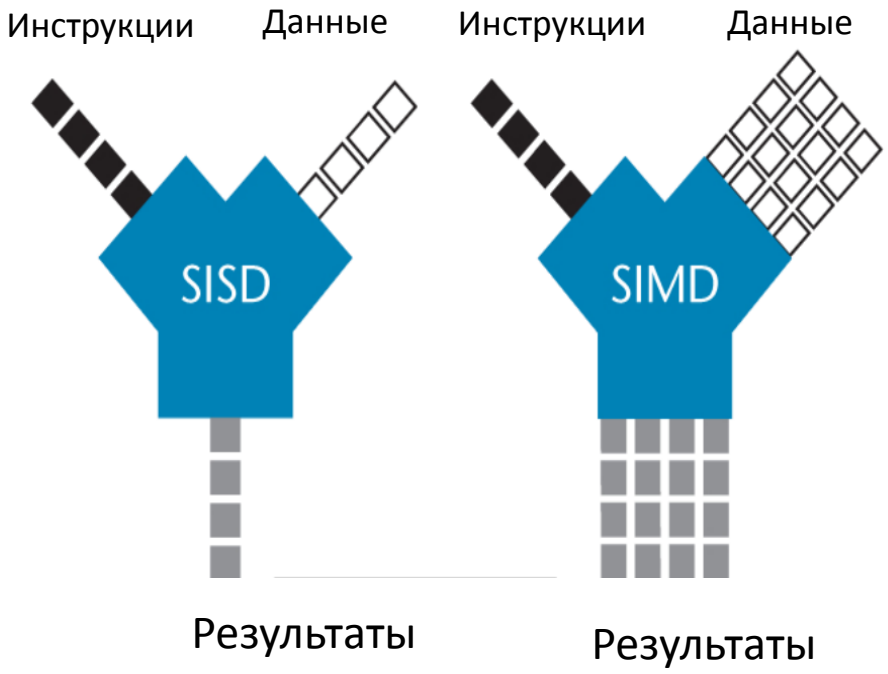
- **Архитектура и возможности векторизации Intel® Xeon Phi™**
- Выравнивание памяти
- Программные способы векторизации
- Интринсики. Типы операций
- Пример умножения матрицы на вектор
- Пример умножения «упакованных» матриц
- Сравнение производительности

Микроархитектура ядра Intel® Xeon Phi™



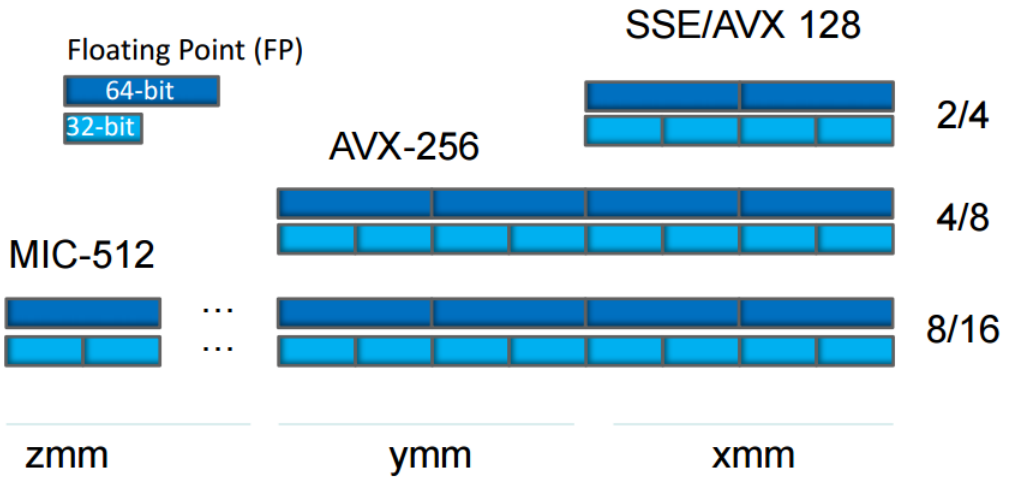
Векторные регистры (SIMD – single instruction multiple data)

- Каждое из 61 ядра имеет модуль векторной обработки (VPU, vector processor unit) – 32 512-разрядных zmm-регистра;
- Одновременные действия над
  16 32-битными целыми/вещественными числами или
  8 64-битными целыми/вещественными числами
- Большинство операций используют 2 аргумента и один результат



X86 SIMD History

Год	Разрядность (бит)	Набор инструкций
1997	64	MMX
1999	128	SSE
2001	128	SSE2
---	128	SSEx (3-4)
2010	256	AVX
2012	512	KNC(Xeon Phi)
2014	256	AVX2
2015	512	AVX-512



- Архитектура и возможности векторизации Intel Xeon Phi
- **Выравнивание памяти**
- Программные способы векторизации
- Интринсики. Типы операций
- Пример умножения матрицы на вектор
- Пример умножения «упакованных» матриц
- Сравнение производительности

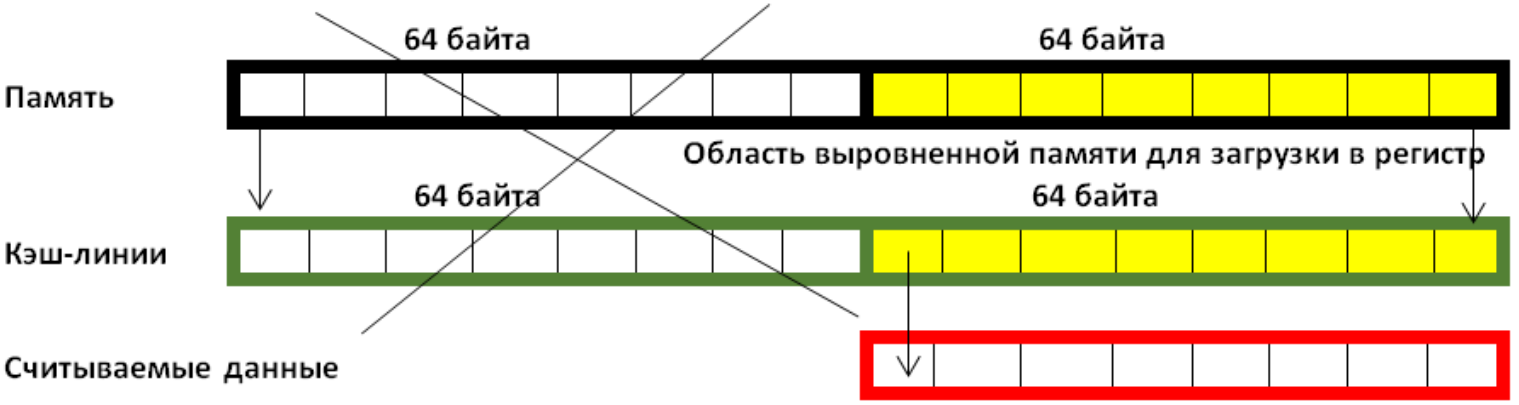
Кэш вычислительного ядра Intel® Xeon Phi™

Уровень КЭШа	L1	L2
Размер	32Kb (инструкции) + 32Kb (данные)	512 Kb (под инструкции и данные)
Ассоциативность/Размер линейки/Число банков	8-way/64 Bytes/8	8-way/64 Bytes/8
Идеальное время доступа	1 такт	11 тактов
Среднее время доступа	3 такта	14-15 тактов
Порт	Чтение или запись	Чтение или запись

Выравнивание памяти

Чтение без выравнивания

Чтение с выравниванием



Выровненное выделение памяти

Статический массив

```
__declspec(align(64)) double p[N]
```

Динамический массив

```
double* p = (double*)_mm_malloc(sz, 64);
_mm_free(p)
```

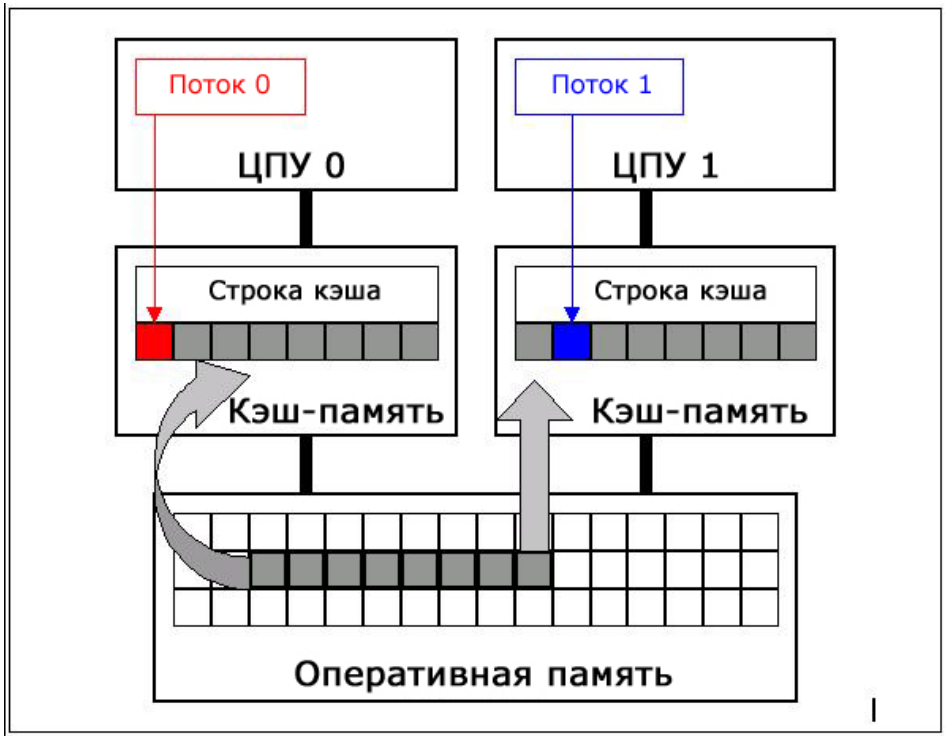
```
#define N 1024
__declspec(align(64)) double
p[N];
double k = 0;
for(int i = 0; i < N; i++)
{
   k += p[i]
}
```

```
#define N 1024
__declspec(align(64)) double
p[N];
double k = 0;
for(int i = 0; i < N; i+=8)
{
  k += p[i]
}
```

Конфликт доступа к КЭШ-памяти

Пример: транспонирование матрицы

Конфликт доступа к КЭШ-памяти в многопоточном режиме

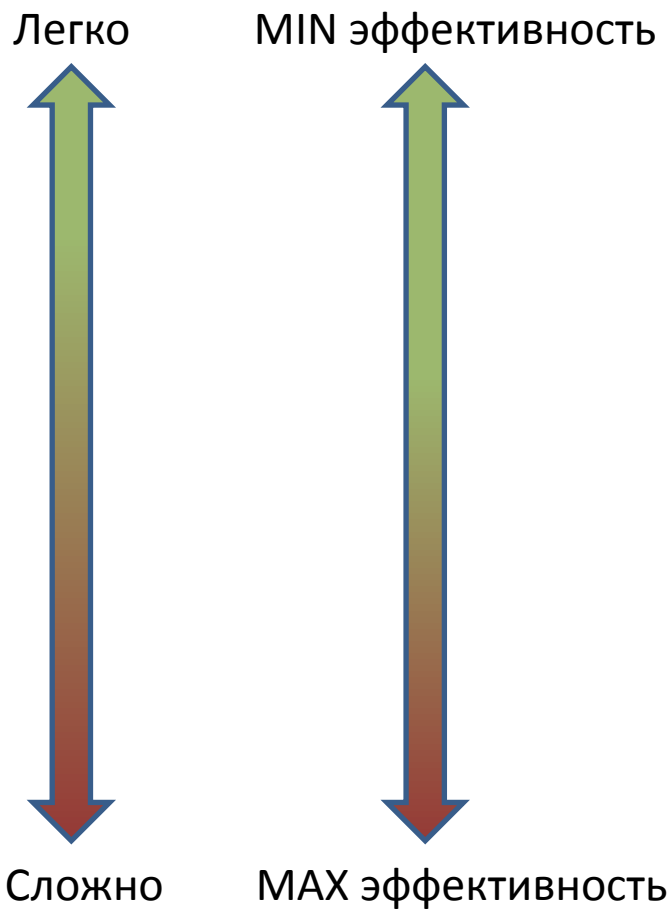



```
double sum=0.0, sum_local[NUM_THREADS];
#pragma omp parallel
num_threads(NUM_THREADS)
{
   int me = omp_get_thread_num();
   sum_local[me]= 0.0;

#pragma omp for
   for (i = 0; i < N; i++)
     sum_local[me] + = x[i] * y[i];

#pragma omp atomicsum += sum_local[me];
}
```

- Архитектура и возможности векторизации Intel Xeon Phi
- Выравнивание памяти
- **Программные способы векторизации**
- Интринсики. Типы операций
- Пример умножения матрицы на вектор
- Пример умножения «упакованных» матриц
- Сравнение производительности

Способы векторизации кода

- Автовекторизация
- Директивы для векторизации
- Intel® Cilk™ Plus Array Notation
- Классы SIMD интринсиков (F64Vec8, F32Vec16)
- Функции-интринсики(__mm512_mul_pd)
- ASSEMBLER (vmulpd zmm {k}, zmm, zmm)



Директивы компилятора

```
//Векторизация с явной длиной вектора 16

#pragma simd vectorlength(16)
for (int i = 0; i < N; i++)
{
    C[i] = B[i] + C[i];
}
```

```
// Векторизация с указанием того, что данные выровнены

#pragma vector aligned
for (int i = 0; i < N; i++)
{
    C[i] = B[i] + C[i];
}
```

```
// Векторизация с игнорированием анализа эффективности и указанием игнорировать зависимости по данным
#pragma ivdep
#pragma vector always
for (int i = 0; i < N; i++)
{
    C[i] = B[i] + C[i];
}
```

Директивы OpenMP

```
void omp_simd (float *A, float *B, float *C)
{
  #pragma omp simd aligned (A, B, C: 64)
  for (int i = 0; i < N; i++)
    C[i] = A[i] + B[i];
}
```

Intel® Cilk™ Plus Array Notation

```
#include <cilk\cilk.h>
void cilk_test(float* a, float* b, float* c)
{
    C[0:n] = A[0:n] + B[0:n];
}
```

Классы SIMD Intrinsic

```
#include <micvec.h>

F64vec8 A[N], B[N], C[N];

for(int i = 0; i < N; i++)
  C[i] = A[i] + B[i];
```

- Архитектура и возможности векторизации Intel Xeon Phi
- Выравнивание памяти
- Программные способы векторизации
- **Интринсики. Типы операций**
- Пример умножения матрицы на вектор
- Пример умножения «упакованных» матриц
- Сравнение производительности

Интринсики. Типы операций

- ***Доступ к данным*:** загрузка и выгрузка данных
- **Математические операции:** сложение, вычитание, умножение, деление, FMA (для вычислений с плавающей запятой)
- ***Вычисление математических функций:*** степень,корень,обращение числа, логарифм, модуль
- ***Логические операции:*** позволяющие выполнять векторные сравнения, находить минимум и максимум и т.д.
- ***Операции редукции***: сложение, умножение, минимум, максимум
- ***Операции перемешивания и объединения данных***

Программирование и компиляция

- Заголовочные файлы для доступа к функциям-интринсикам для различных наборов SIMD инструкций

```
#include <mmintrin.h>    // MMX
#include <xmmintrin.h>  //SSE
#include <emmintrin.h> //SSE2
#include <pmmintrin.h> //SSE3
#include <tmmintrin.h>  //SSSE3
#include <smmintrin.h> //SSE4
#include <avxintrin.h>    //AVX
#include <fmaintrin.h>   //FMA
```

#include <immintrin.h> // KNC и другие

Компилятор Intel, команда для компиляции

icc –O2 -mmic test.cpp -std=c++11 -o test.exe
scp test.exe mic0:~
ssh mic0
./test.exe

Интринсики. Типы данных

__m512: 16 x float
__m512d: 8 x double
__m512i: 16 x int32 или 8 x int64
__mmask8: 8-битная маска
__mmask16: 16-битная маска

Префикс интринсиков для Intel® Xeon Phi™: _mm512_
Многие операции имеют суффикс для обозначения типов данных:

_mm512_load_ps: загрузка float-вектора
_mm512_load_pd: загрузка double-вектора
_mm512_load_epi32: загрузка целых 32-разрядных чисел
_mm512_load_epi64: загрузка целых 64-разрядных чисел

Операции загрузки и выгрузки данных

- **Загрузка**

__m512d _mm512_load_pd (void const* mt)

__m512d _mm512_extload_pd (void const* mt,

_MM_UPCONV_PD_ENUM conv, _MM_BROADCAST64_ENUM bc, int hint)

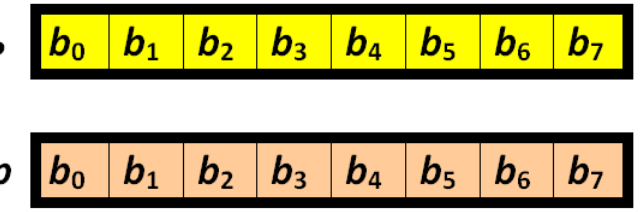
- **Выгрузка**

void _mm512_mask_store_pd (void* mem_addr, __mmask8 k, __m512d a)

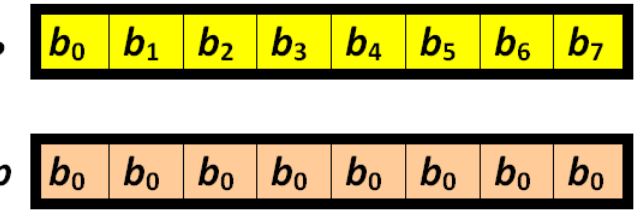
void _mm512_store_pd (void* mem_addr, __m512d a)

_mm512_extload_pd
_MM_BROADCAST64_ ENUM

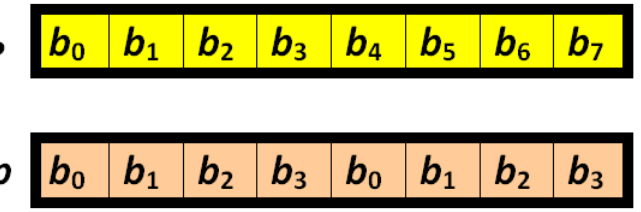
- _MM_BROADCAST64_NONE



- _MM_BROADCAST_1X8



- _MM_BROADCAST_4X8



Инструкции для математических функций

Ускоренное вычисление 2^N (только для одинарной точности)

__m512 _mm512_exp223_ps (__m512i a)

Ускоренное вычисление $\log_2$ (только для одинарной точности)

__m512 _mm512_log2_ps (__m512 a)

Ускоренное обращение числа

__m512 _mm512_rcp23_ps (__m512 a)

Ускоренное вычисление квадратного корня

__m512 _mm512_rsqrt23_ps (__m512 a)

Вычисление модуля для одинарной точности

__m512 _mm512_abs_ps (__m512 a)

Вычисление модуля для двойной точности

__m512d _mm512_abs_pd (__m512d a)

Пакетные арифметические инструкции

Сложение

__m512d _mm512_add_pd (__m512d a, __m512d b)

Умножение

__m512d _mm512_mul_pd (__m512d a, __m512d b)

Сложение с отрицанием

__m512d _mm512_addn_pd (__m512d a, __m512d b)

FMA – инструкции (смешанное умножение со сложением)

Смешанное умножение со сложением:

d = a*b+c

__m512d _mm512_fmadd_pd (__m512d a, __m512d b, __m512d c)

Смешанное умножение с вычитанием:

d = a*b-c

__m512d _mm512_fmsub_pd (__m512d a, __m512d b, __m512d c)

Операции редукции

- Вычисление минимального числа

  double _mm512_reduce_min_pd (__m512d a)
- Вычисление максимального числа

  double _mm512_reduce_max_pd (__m512d a)
- Сложение элементов в регистре

  double _mm512_reduce_add_pd (__m512d a)
- Умножение элементов в регистре

  double _mm512_reduce_mul_pd (__m512d a)

Операции сравнения

- Проверка на меньше, либо равно

__mmask8 _mm512_cmple_pd_mask (__m512d a, __m512d b)

- Проверка на строго больше

__mmask8 _mm512_cmpnle_pd_mask (__m512d a, __m512d b)

- Сравнение на равенство

__mmask8 _mm512_cmpeq_pd_mask (__m512d a, __m512d b)

- Сравнение на неравенство

__mmask8 _mm512_cmpneq_pd_mask (__m512d a, __m512d b)

Операции перемешивания элементов

Смешивание 2 регистров по маске

__m512d _mm512_mask_blend_pd (__mmask8 k, __m512d a, __m512d b)

Перестановка элементов в 4-элементных частях регистра

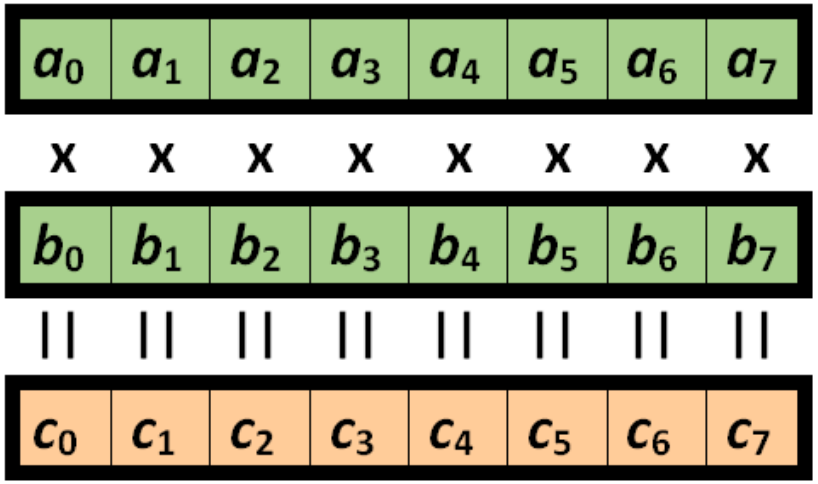
__m512d _mm512_swizzle_pd (__m512d v, _MM_SWIZZLE_ENUM s)

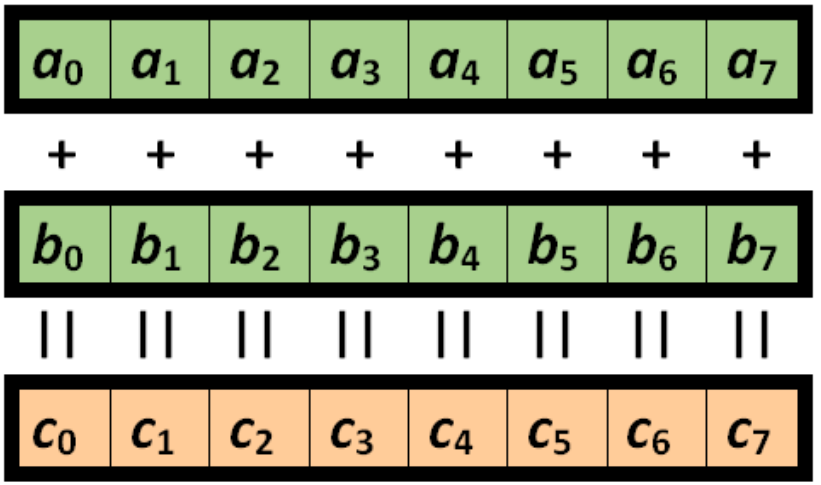
_MM_SWIZZLE_ENUM:

_MM_SWIZ_REG_NONE - hgfe dcba
_MM_SWIZ_REG_DCBA - hgfe dcba
_MM_SWIZ_REG_CDAB - ghef cdab
_MM_SWIZ_REG_BADC - fehg badc
_MM_SWIZ_REG_AAAA - eeee aaaa
_MM_SWIZ_REG_BBBB - ffff bbbb
_MM_SWIZ_REG_CCCC - gggg cccc
_MM_SWIZ_REG_DDDD - hhhh dddd
_MM_SWIZ_REG_DACB - hfeg dbac

Пакетные операции с двумя операторами

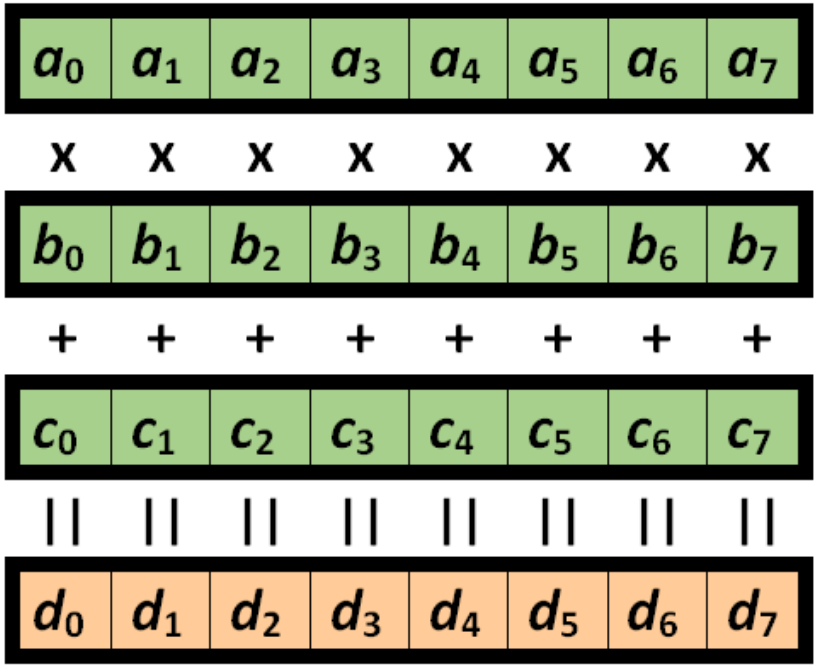
__m512d _mm512_mul_pd (__m512d a, __m512d b)
__m512d _mm512_add_pd (__m512d a, __m512d b)





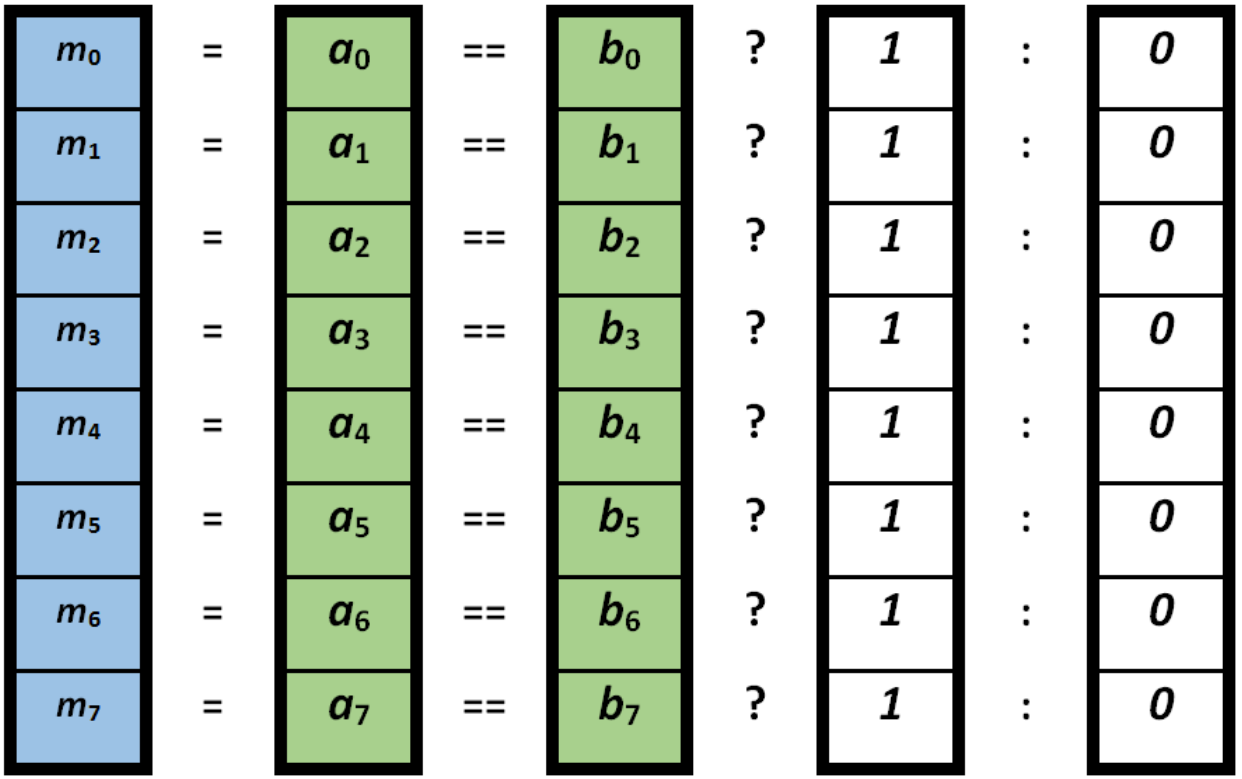
Пакетные операции с тремя операторами

__m512d _mm512_fmadd_pd (__m512d a, __m512d b, __m512d c)
__m512d _mm512_fmsub_pd (__m512d a, __m512d b, __m512d c)

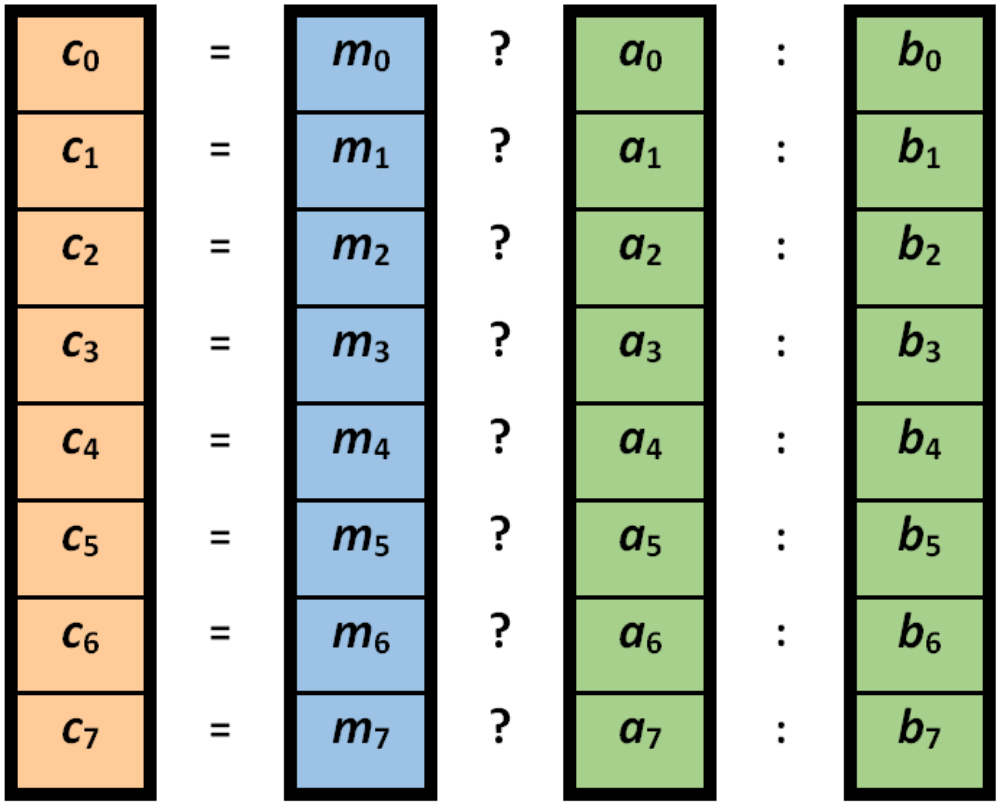


Операции сравнения

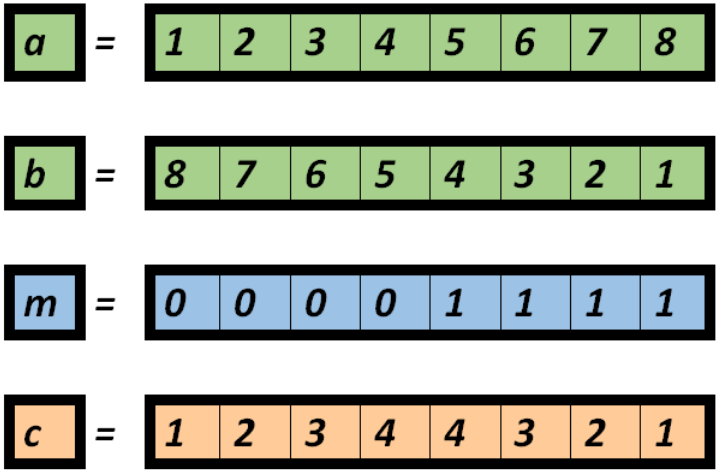
__mmask8 _mm512_cmpeq_pd_mask (__m512d a, __m512d b)



__m512d _mm512_mask_blend_pd (__mmask8 m, __m512d a, __m512d b)



Пример



- Архитектура и возможности векторизации Intel Xeon Phi
- Выравнивание памяти
- Программные способы векторизации
- Интринсики. Типы операций
- **Пример умножения матрицы на вектор**
- Пример умножения «упакованных» матриц
- Сравнение производительности

Умножение матрицы на вектор

$$\mathrm{b} = A^T \times a$$

$$\begin{aligned}
b_0 &= A_{00}a_0 + A_{10}a_1 + A_{20}a_2 + A_{30}a_3 \\
b_1 &= A_{01}a_0 + A_{11}a_1 + A_{21}a_2 + A_{31}a_3 \\
b_2 &= A_{02}a_0 + A_{12}a_1 + A_{22}a_2 + A_{32}a_3 \\
b_3 &= A_{03}a_0 + A_{13}a_1 + A_{23}a_2 + A_{33}a_3
\end{aligned}$$

Хранение матрицы по строкам:

$$A = [A_{00}, A_{01}, A_{02}, A_{03}, A_{10}, A_{11}, A_{12}, A_{13}, A_{20}, A_{21}, A_{22}, A_{23}, A_{30}, A_{31}, A_{32}, A_{33}]$$

Код загрузки данных в регистры для матрицы 16x16

```
zmm00 = _mm512_load_ps(mtx);
zmm01 = _mm512_load_ps(mtx + VECTOR_SIZE);
zmm02 = _mm512_load_ps(mtx + VECTOR_SIZE * 2);
zmm03 = _mm512_load_ps(mtx + VECTOR_SIZE * 3);
zmm04 = _mm512_load_ps(mtx + VECTOR_SIZE * 4);
zmm05 = _mm512_load_ps(mtx + VECTOR_SIZE * 5);
zmm06 = _mm512_load_ps(mtx + VECTOR_SIZE * 6);
zmm07 = _mm512_load_ps(mtx + VECTOR_SIZE * 7);
zmm08 = _mm512_load_ps(mtx + VECTOR_SIZE * 8);
zmm09 = _mm512_load_ps(mtx + VECTOR_SIZE * 9);
zmm10 = _mm512_load_ps(mtx + VECTOR_SIZE * 10);
zmm11 = _mm512_load_ps(mtx + VECTOR_SIZE * 11);
zmm12 = _mm512_load_ps(mtx + VECTOR_SIZE * 12);
zmm13 = _mm512_load_ps(mtx + VECTOR_SIZE * 13);
zmm14 = _mm512_load_ps(mtx + VECTOR_SIZE * 14);
zmm15 = _mm512_load_ps(mtx + VECTOR_SIZE * 15);
...
```

```
…
zmm16 = _mm512_extload_ps(vec, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm17 = _mm512_extload_ps(vec + 1, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm18 = _mm512_extload_ps(vec + 2, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm19 = _mm512_extload_ps(vec + 3, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm20 = _mm512_extload_ps(vec + 4, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm21 = _mm512_extload_ps(vec + 5, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm22 = _mm512_extload_ps(vec + 6, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm23 = _mm512_extload_ps(vec + 7, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm24 = _mm512_extload_ps(vec + 8, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm25 = _mm512_extload_ps(vec + 9, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm26 = _mm512_extload_ps(vec + 10, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm27 = _mm512_extload_ps(vec + 11, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm28 = _mm512_extload_ps(vec + 12, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm29 = _mm512_extload_ps(vec + 13, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm30 = _mm512_extload_ps(vec + 14, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm31 = _mm512_extload_ps(vec + 15, _MM_UPCONV_PS_NONE, _MM_BROADCAST_1X16, _MM_HINT_NONE);
```

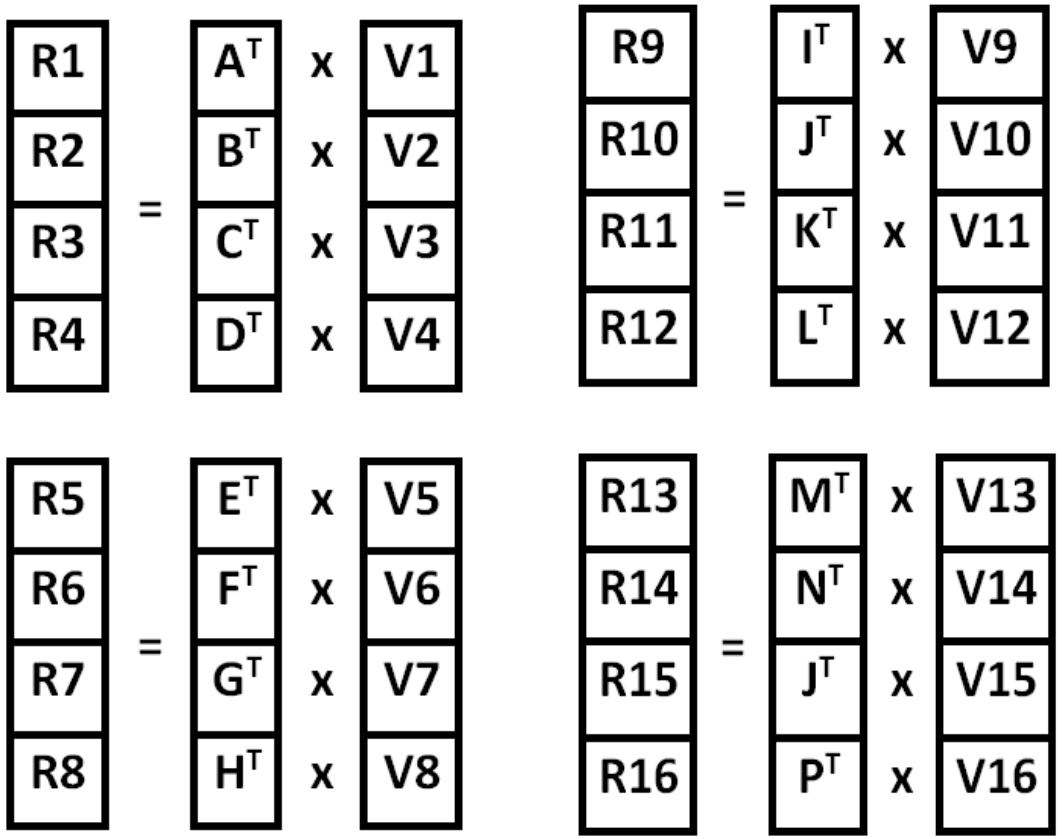
Код вычисления и выгрузки результата

```
zmm00 = _mm512_mul_ps(zmm00, zmm16);
zmm00 = _mm512_fmadd_ps(zmm01, zmm17, zmm00);
zmm00 = _mm512_fmadd_ps(zmm02, zmm18, zmm00);
zmm00 = _mm512_fmadd_ps(zmm03, zmm19, zmm00);
zmm00 = _mm512_fmadd_ps(zmm04, zmm20, zmm00);
zmm00 = _mm512_fmadd_ps(zmm05, zmm21, zmm00);
zmm00 = _mm512_fmadd_ps(zmm06, zmm22, zmm00);
zmm00 = _mm512_fmadd_ps(zmm07, zmm23, zmm00);
zmm00 = _mm512_fmadd_ps(zmm08, zmm24, zmm00);
zmm00 = _mm512_fmadd_ps(zmm09, zmm25, zmm00);
zmm00 = _mm512_fmadd_ps(zmm10, zmm26, zmm00);
zmm00 = _mm512_fmadd_ps(zmm11, zmm27, zmm00);
zmm00 = _mm512_fmadd_ps(zmm12, zmm28, zmm00);
zmm00 = _mm512_fmadd_ps(zmm13, zmm29, zmm00);
zmm00 = _mm512_fmadd_ps(zmm14, zmm30, zmm00);
zmm00 = _mm512_fmadd_ps(zmm15, zmm31, zmm00);
_mm512_store_ps(res, zmm00);
```

- Архитектура и возможности векторизации Intel Xeon Phi
- Выравнивание памяти
- Программные способы векторизации
- Интринсики. Типы операций
- Пример умножения матрицы на вектор
- **Пример умножения «упакованных» матриц**
- Сравнение производительности

Умножение упакованных матриц на вектора

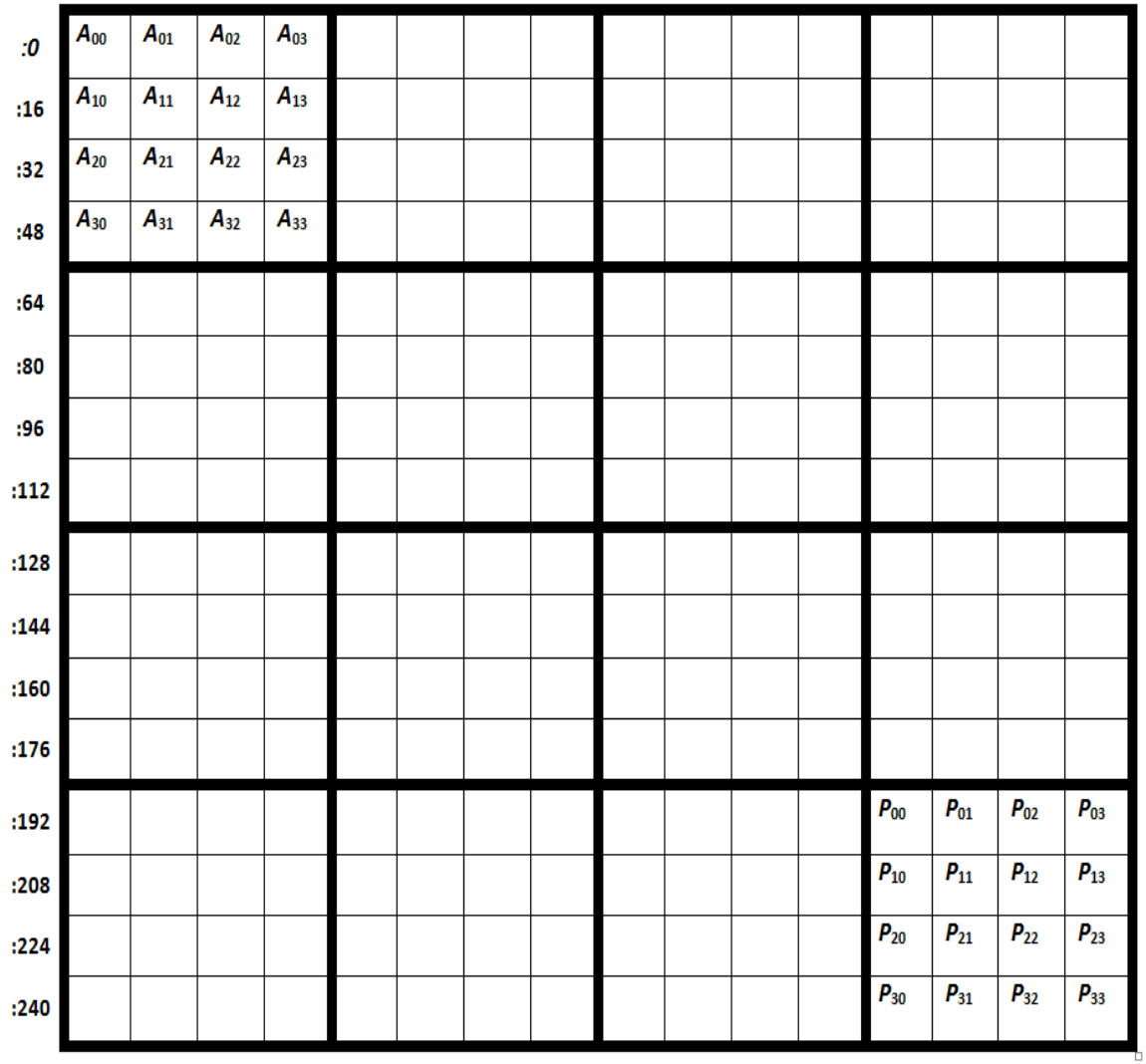
Умножение матриц 4x4 на вектора 4x1



Упаковка матриц 4x4 в матрицу 16x16

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Расположение элементов в памяти



- Элементы векторов хранятся последовательно
- Строки матриц хранятся по порядку для каждых 4 матриц: сначала четыре первые строки, потом вторые, третьи и четвертые

Код загрузки данных

```
const int N = 4;
const int mask32_0 = 0x00F0;
const int mask32_1 = 0xF000;
const int mask32_2 = 0xFF00;
for (int i = 0; i < N; i++)
{
REAL* vec = invec + i * VECTOR_SIZE;
REAL* mtx = inmtx + i * VECTOR_SIZE * N;
zmm00 = _mm512_load_ps(inmtx);
zmm01 = _mm512_load_ps(inmtx + VECTOR_SIZE);
zmm02 = _mm512_load_ps(inmtx + VECTOR_SIZE * 2);
zmm03 = _mm512_load_ps(inmtx + VECTOR_SIZE * 3);
...
```

```
...
 zmm08 = _mm512_extload_ps(vec + 0, _MM_UPCONV_PS_NONE,
                _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm09 = _mm512_extload_ps(vec + 4, _MM_UPCONV_PS_NONE,
                _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm10 = _mm512_extload_ps(vec + 8, _MM_UPCONV_PS_NONE,
                _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm11 = _mm512_extload_ps(vec + 12, _MM_UPCONV_PS_NONE,
                _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm12 = _mm512_extload_ps(vec + 1, _MM_UPCONV_PS_NONE,
                _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm13 = _mm512_extload_ps(vec + 5, _MM_UPCONV_PS_NONE,
                _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm14 = _mm512_extload_ps(vec + 9, _MM_UPCONV_PS_NONE,
                _MM_BROADCAST_1X16, _MM_HINT_NONE);
zmm15 = _mm512_extload_ps(vec + 13, _MM_UPCONV_PS_NONE,
                _MM_BROADCAST_1X16, _MM_HINT_NONE);

zmm06 = _mm512_mask_blend_ps(mask32_0, zmm08, zmm09);
zmm07 = _mm512_mask_blend_ps(mask32_1, zmm10, zmm11);
zmm04 = _mm512_mask_blend_ps(mask32_2, zmm06, zmm07);
zmm06 = _mm512_mask_blend_ps(mask32_0, zmm12, zmm13);
zmm07 = _mm512_mask_blend_ps(mask32_1, zmm14, zmm15);
zmm05 = _mm512_mask_blend_ps(mask32_2, zmm06, zmm07);
...
}
```

Код вычисления и выгрузки результатов

```
for (int i = 0; i < N; i++)
{
            …. Код загрузки и подготовки даннных …
      zmm16 = _mm512_mul_ps(zmm00, zmm04);
      zmm16 = _mm512_fmadd_ps(zmm01, zmm05, zmm16);
      zmm16 = _mm512_fmadd_ps(zmm02, zmm06, zmm16);
      zmm16 = _mm512_fmadd_ps(zmm03, zmm07, zmm16);

      _mm512_store_ps(res+i*16, zmm16);
}
```

Оптимальная версия кода с использованием SWIZZLE

```
const int N = 4;
for (int i = 0; i < N; i++)
{
        REAL* vec = invec + i * VECTOR_SIZE;
        REAL* mtx = inmtx + i * VECTOR_SIZE * N;
        zmm00 = _mm512_load_ps(inmtx);
        zmm01 = _mm512_load_ps(inmtx + VECTOR_SIZE);
        zmm02 = _mm512_load_ps(inmtx + VECTOR_SIZE * 2);
        zmm03 = _mm512_load_ps(inmtx + VECTOR_SIZE * 3);
        zmm08 = _mm512_load_ps(vec);
        zmm04 = _mm512_swizzle_ps(zmm08, _MM_SWIZ_REG_AAAA);
        zmm05 = _mm512_swizzle_ps(zmm08, _MM_SWIZ_REG_BBBB);
        zmm06 = _mm512_swizzle_ps(zmm08, _MM_SWIZ_REG_CCCC);
        zmm07 = _mm512_swizzle_ps(zmm08, _MM_SWIZ_REG_DDDD);
        zmm16 = _mm512_mul_ps(zmm00, zmm04);
        zmm16 = _mm512_fmadd_ps(zmm01, zmm05, zmm16);
        zmm16 = _mm512_fmadd_ps(zmm02, zmm06, zmm16);
        zmm16 = _mm512_fmadd_ps(zmm03, zmm07, zmm16);
        _mm512_store_ps(res+i*16, zmm16);
}
```

- Архитектура и возможности векторизации Intel Xeon Phi
- Выравнивание памяти
- Программные способы векторизации
- Интринсики. Типы операций
- Пример умножения матрицы на вектор
- Пример умножения «упакованных» матриц
- **Сравнение производительности**

Сравнение производительности

– Умножение матриц на вектор

	t auto	t auto loop inv	t simd	s loop inv	s simd
float	4.16	1	0.75	4.16	**5.55**
double	5.77	3.1	1.6	1.86	**3.61**

– Умножение упакованных матриц на вектор

	t auto	t simd blend	t simd swizzle	s simd blend	s simd swizzle
float	1.15	0.34	0.2	3.38	**5.75**
double	1.41	0.55	0.45	2.56	**3.13**

Основные выводы

- Векторизация кода с помощью функций-интринсиков позволяет в ряде случаев получить выигрыш производительности по сравнению с автоматической векторизацией
- Ручная векторизация требует особой организации данных в памяти и использования выровненной памяти
- Количество регистров ограничено, но повторная загрузка данных в регистры замедляет программу, поэтому надо использовать максимально доступное число регистров (32 для Intel Xeon Phi)
- При работе в многопоточном режиме необходимо следить за отсутствием конфликтов доступа к кэш-памяти и разделять задачу по расположению данных в памяти

Ссылки

Achieving Superior Performance on Black-Scholes Valuation Computing using Intel® Xeon Phi™ Coprocessors

https://software.intel.com/en-us/articles/case-study-achieving-superior-performance-on-black-scholes-valuation-computing-using

Intel® Xeon Phi™ Core Micro-architecture

https://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture

Intel® Intrinsic Guide

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

Спасибо за внимание

http://www.singularis-lab.com/

SINGULARIS LAB

software development