



Оптимизация производительности Postgres с применением векторной обработки массивов данных

Артем Бугаенко, разработчик «Тантор Лабс»



Подготовка окружения



Все необходимые примеры лежат в папке **AVX2**, там же лежит файл **commands.txt**, в котором описан список действий и представлены необходимые команды.

<https://github.com/TantorLabs/meetups>



Обо мне



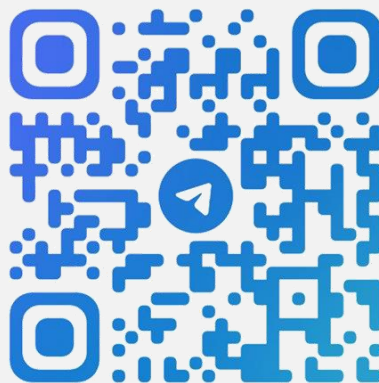
<https://github.com/TantorLabs/meetups>

Образование:

- Магистратура СПб ГУАП

Опыт работы:

- Константа (УЗК)
- Контур НИИРС (ПАК)
- SK Hynix (Enterprise SSD)
- Tantor (PostgreSQL)



@BUGGAI



Что мы сегодня рассмотрим



- Краткий экскурс в SIMD
- Преимущества и ограничения SIMD
- Знакомство с SIMD intrinsics
- Разбор примеров применения SIMD
- Подводные камни
- Использование SIMD в PostgreSQL
- Перспективные направления для оптимизации
- Вопросы и ответы



Что такое **SIMD** (и SISD)?



SISD – (Single Instruction Stream & Single Data Stream),

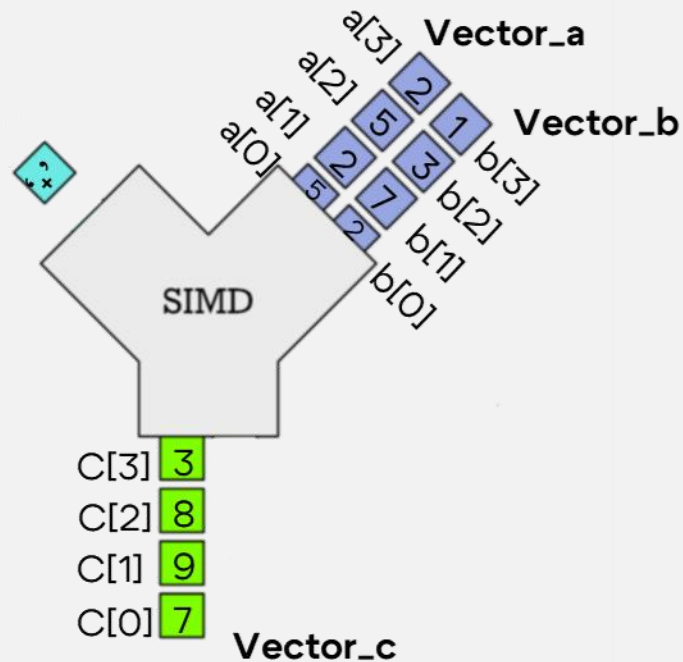
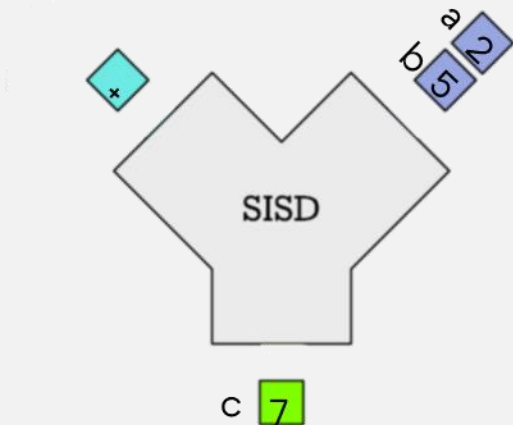
или **ОКОД** (Одиночный поток Команд и Одиночный поток Данных) – архитектура компьютера, в которой один процессор выполняет один поток команд, оперируя одним потоком данных.

SIMD – (Single Instruction Stream & Multiple Data Stream),

или **ОКМД** (Одиночный поток Команд и Множественный поток Данных) – архитектура, в которой есть возможность выполнять одну арифметическую операцию сразу над многими данными – элементами вектора.



Идеологии SIMD и SISD



Инструкции



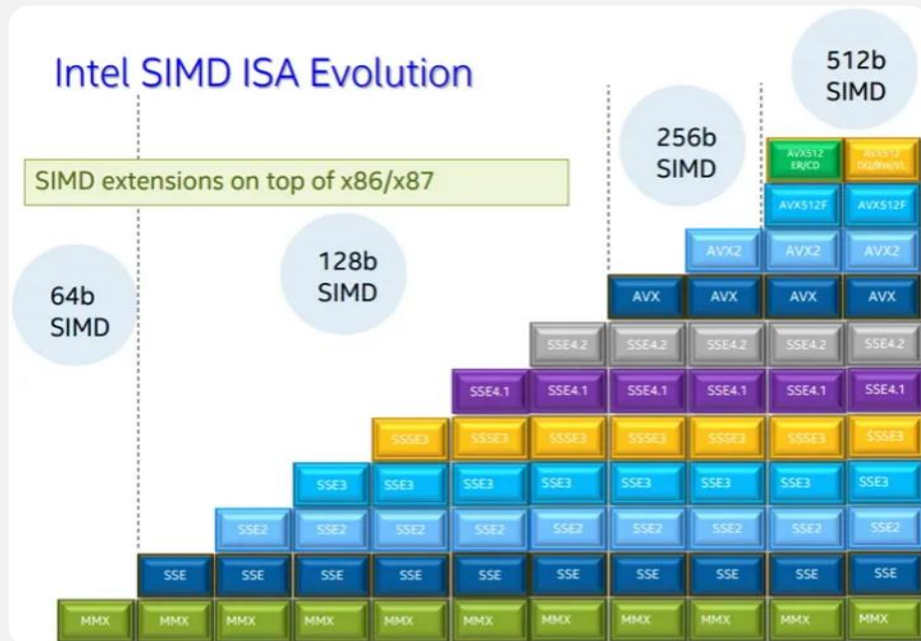
Данные



Результат

Как всё начиналось?

Пирамида развития SIMD на CPU



Основные вехи и года их появления:

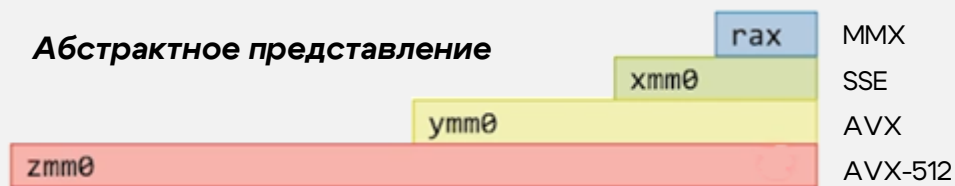
- MMX 1997 г. (Intel Pentium MMX)
- SSE 1999 г. (Intel Pentium III)
- SSE4.2 2008 г. (Intel Core i7)
- AVX 2011 г. (Intel Sandy Bridge)
- AVX2 2013 г. (Intel Haswell)
- AVX512F 2016 г. (Intel Xeon Phi и Intel Skylake-SP)

AMD:

- MMX 1997 г. (AMD K6-2)
- 3DNow! 1998 г. (AMD K6-2)
- SSE 2001 г. (AMD Athlon XP)
- SSE2 2003 г. (AMD Athlon 64)
- SSE3 2005 г. (AMD Athlon 64 Venice/San Diego)
- SSE4a 2007 г. (AMD Phenom)
- SSE4.1, SSE4.2 2011 г. (AMD Bulldozer)
- AVX 2011 г. (AMD Bulldozer)
- AVX2 2015 г. (AMD Excavator)
- AVX-512 2022 г. (AMD Zen 4, серия Ryzen 7000)

Версии SIMD и их отличия

- Расширения SSE и его последующие версии добавили регистры xmm0-xmm15, отдельные от старого стека, размером 128 бит
- Расширения AVX добавили ymm и zmm регистры размером 256 и 512 бит соответственно, и инструкции для работы с ними



AVX-512 register scheme as extension from the AVX (YMM0-YMM15) and SSE (XMM0-XMM15) registers

511	256	255	128	127	0
ZMM0	YMM0	XMM0			
ZMM1	YMM1	XMM1			
ZMM2	YMM2	XMM2			
ZMM3	YMM3	XMM3			
ZMM4	YMM4	XMM4			
ZMM5	YMM5	XMM5			
ZMM6	YMM6	XMM6			
ZMM7	YMM7	XMM7			
ZMM8	YMM8	XMM8			
ZMM9	YMM9	XMM9			
ZMM10	YMM10	XMM10			
ZMM11	YMM11	XMM11			
ZMM12	YMM12	XMM12			
ZMM13	YMM13	XMM13			
ZMM14	YMM14	XMM14			
ZMM15	YMM15	XMM15			
ZMM16	YMM16	XMM16			
ZMM17	YMM17	XMM17			
ZMM18	YMM18	XMM18			
ZMM19	YMM19	XMM19			
ZMM20	YMM20	XMM20			
ZMM21	YMM21	XMM21			
ZMM22	YMM22	XMM22			
ZMM23	YMM23	XMM23			
ZMM24	YMM24	XMM24			
ZMM25	YMM25	XMM25			
ZMM26	YMM26	XMM26			
ZMM27	YMM27	XMM27			
ZMM28	YMM28	XMM28			
ZMM29	YMM29	XMM29			
ZMM30	YMM30	XMM30			
ZMM31	YMM31	XMM31			

**Представление в процессоре
с поддержкой AVX-512**

Что умеет SIMD от MMX к AVX



SSE (SSE1, 1999) (70 инструкций)

- Арифметика с плавающей точкой:
- Логические операции:
- Сравнение:
- Перестановка данных:

ADDPS, SUBPS, MULPS, DIVPS, SQRTPS
ANDPS, ORPS, XORPS
CMPPS
SHUFPS, MOVAPS

SSE2 (2001) (214 инструкций)

- Арифметика с целыми числами:
- Дополнительно с плавающей точкой:
- Перемещение данных:

PADDB, PSUBB, PMULHW
ADDPD, MULPD, SQRTPD
MOVDQA

SSE3 (2004) (227 инструкций)

- Горизонтальные операции:
- Дублирование:

HADDPS, HSUBPS
MOVSLDUP

SSSE3 (2006) (259 инструкций)

- Модификация знака:
- Перестановка байт:

PSIGNB, PABSB
PSHUFB

SSE4.1 (2007) (306 инструкций)

- Операции с целыми числами:
- Округление:
- Скалярное произведение: DPPS

PMULLD, PTEST
ROUNDPS

SSE4.2 (2008) (313 инструкций)

- Сравнение строк:
- CRC32:

PCMPISTRI
CRC32



Что сейчас актуально?

Схема различий последних версий SIMD

Intel® AVX	Intel® AVX2
128/256-bit FP 16 registers NDS (and AVX128) Improved blend MASKMOV Implicit unaligned	Float16 128/256-bit FP FMA 256-bit int PERMD Gather

Intel® AVX-512
128/256/512-bit FP/Int 32 vector registers 8 mask registers 512-bit embedded rounding Embedded broadcast Scalar/SSE/AVX “promotions” Native media additions HPC additions Transcendental support Gather/Scatter Flag-based enumeration Intel® Xeon P-core only

Intel® AVX10.1 (pre-enabling)
Optional 512-bit FP/Int 128/256-bit FP/Int 32 vector registers 8 mask registers 512-bit embedded rounding Embedded broadcast Scalar/SSE/AVX “promotions” Native media additions HPC additions Transcendental support Gather/Scatter Version-based enumeration Intel® Xeon P-core only

Intel® AVX10.2
New data movement, transforms and type instructions Optional 512-bit FP/Int 128/256-bit FP/Int 32 vector registers 8 mask registers 256/512-bit embedded rounding Embedded broadcast Scalar/SSE/AVX “promotions” Native media additions HPC additions Transcendental support Gather/Scatter Version-based enumeration Supported on P-cores, E-cores

С чего начать



1. Поддержка процессором
2. Подключение библиотек
3. Флаги компиляции




Доступность SIMD



Использование сторонних программ

HWiNFO64 v7.04-4480 @ ASUS System Product Name - System S

CPU			
	Intel Core i9-9900K		14 nm
Stepping	P0	Cores/Threads	8 / 16
Codename	Coffee Lake-S	µCU	D6
SSPEC	SRELS, SRESB	Prod. Unit	
CPU #0	Platform Socket H4 (LGA1151)		
TDP	95 W	Cache	8x32 + 8x32 + 8x256 + 16M
Features			
MMX	3DNow!	3DNow!-2	SSE SSE-2 SSE-3 SSE-3
SSE4A	SSE4.1	SSE4.2	AVX AVX2 AVX-512
BMI2	ABM	TBM	FMA ADX XOP
DEP	VMX	SMX	SMEP SMAP TSX MPX
EM64T	EIST	TM1	TM2 HTT Turbo SST
AES-NI	RDRAND	RDSEED	SHA SGX TME

Средствами Linux

```
cat /proc/cpuinfo | grep -m 1 -o -E  
'sse|sse2|sse3|sse3|sse4_1|sse4_2|avx|avx2|avx512f' | sort | tr  
' ' '\n'
```

Средствами C

```
1#include <stdio.h>  
2  
3void cpuid(int info[4], int InfoType, int Subleaf) {  
4    __asm__ __volatile__ (  
5        "cpuid" :  
6        "=a" (info[0]), "=b" (info[1]), "=c" (info[2]), "=d" (info[3]) :  
7        "a" (InfoType), "c" (Subleaf) // Передаем и InfoType, и Subleaf  
8    );  
9}  
10  
11int main() {  
12    int info[4];  
13  
14    // Получаем информацию о поддержке инструкций  
15    cpuid(info, 1, 0);  
16    int ecx = info[2];  
17    int edx = info[3];  
18  
19    cpuid(info, 7, 0);  
20    int ebx = info[1];  
21    // Проверка поддержки различных инструкций  
22    if (edx & (1 << 25)) printf("SSE supported\n");  
23    if (edx & (1 << 26)) printf("SSE2 supported\n");  
24    if (ecx & (1 << 0)) printf("SSE3 supported\n");  
25    if (ecx & (1 << 9)) printf("SSSE3 supported\n");  
26    if (ecx & (1 << 19)) printf("SSE4.1 supported\n");  
27    if (ecx & (1 << 20)) printf("SSE4.2 supported\n");  
28    if (ecx & (1 << 28)) printf("AVX supported\n");  
29    if (ebx & (1 << 5)) printf("AVX2 supported\n");  
30    if (ebx & (1 << 16)) printf("AVX-512F supported\n");  
31  
32    return 0;  
33}
```



Доступность SIMD



Команда и пример выполнения программы

```
$ gcc -O2 -o supp 01_support.c
$ ./supp
SSE supported
SSE2 supported
SSE3 supported
SSSE3 supported
SSE4.1 supported
SSE4.2 supported
AVX supported
AVX2 supported
$
```

Исходный код

```
1#include <stdio.h>
2
3void cpuid(int info[4], int InfoType, int Subleaf) {
4    __asm__ __volatile__ (
5        "cpuid" :
6        "=a" (info[0]), "=b" (info[1]), "=c" (info[2]), "=d" (info[3]) :
7        "a" (InfoType), "c" (Subleaf) // Передаем и InfoType, и Subleaf
8    );
9}
10
11int main() {
12    int info[4];
13
14    // Получаем информацию о поддержке инструкций
15    cpuid(info, 1, 0);
16    int ecx = info[2];
17    int edx = info[3];
18
19    cpuid(info, 7, 0);
20    int ebx = info[1];
21    // Проверка поддержки различных инструкций
22    if (edx & (1 << 25)) printf("SSE supported\n");
23    if (edx & (1 << 26)) printf("SSE2 supported\n");
24    if (ecx & (1 << 0)) printf("SSE3 supported\n");
25    if (ecx & (1 << 9)) printf("SSSE3 supported\n");
26    if (ecx & (1 << 19)) printf("SSE4.1 supported\n");
27    if (ecx & (1 << 20)) printf("SSE4.2 supported\n");
28    if (ecx & (1 << 28)) printf("AVX supported\n");
29    if (ebx & (1 << 5)) printf("AVX2 supported\n");
30    if (ebx & (1 << 16)) printf("AVX-512F supported\n");
31
32    return 0;
33}
```



Как подключить



Раньше для каждой версии SIMD интринсики подключались из разных файлов:

```
#include <mmintrin.h> // MMX
#include <xmmintrin.h> // SSE
#include <emmintrin.h> // SSE2
#include <pmmmintrin.h> // SSE3
#include <smmmintrin.h> // SSE4.1
#include <nmmmintrin.h> // SSE4.2
#include <ammintrin.h> // SSE4A
#include <wmmmintrin.h> // AES
#include <immintrin.h> // AVX, AVX2, AVX-512F, FMA
```

**С появлением AVX
достаточно подключить
<immintrin.h>.**

Интринсик (от англ. **intrinsic**) — это функция, при вызове которой компилятор генерирует конкретные аппаратные инструкции.



Типы данных

SSE (`__m128`, `__m128i`, `__m128d`): Эти типы используются для операций на 128-битных регистрах, позволяя обработать четыре 32-битных числа или два 64-битных числа одновременно.

AVX2 (`__m256`, `__m256i`, `__m256d`): Эти типы позволяют работать с 256-битными регистрами, что удваивает количество обрабатываемых данных по сравнению с SSE.

С младшей половиной `ymm` регистра SSE может работать, как с полноценным `xmm` регистром.

Регистры AVX + SSE

255	128	127	0
YMM0	XMM0		
YMM1	XMM1		
YMM2	XMM2		
YMM3	XMM3		
YMM4	XMM4		
YMM5	XMM5		
YMM6	XMM6		
YMM7	XMM7		
YMM8	XMM8		
YMM9	XMM9		
YMM10	XMM10		
YMM11	XMM11		
YMM12	XMM12		
YMM13	XMM13		
YMM14	XMM14		
YMM15	XMM15		

Установка вектора



Интринсики установки вектора

```
// Загрузка данных в AVX2-регистры
// 32-битные числа с плавающей точкой (одинарная точность)
__m256 vec_float1 = _mm256_loadu_ps(float_data1);    // Невыровненные данные
__m256 vec_float2 = _mm256_load_ps(float_data2);      // Выровненные данные
// 16-битные целые числа
__m256i vec_int16_1 = _mm256_loadu_si256((__m256i*)int_data1_16); // Невыровненные данные
__m256i vec_int16_2 = _mm256_load_si256((__m256i*)int_data2_16); // Выровненные данные
// 32-битные целые числа
__m256i vec_int32_1 = _mm256_loadu_si256((__m256i*)int_data1_32); // Невыровненные данные
__m256i vec_int32_2 = _mm256_load_si256((__m256i*)int_data2_32); // Выровненные данные
// 64-битные целые числа
__m256i vec_int64_1 = _mm256_loadu_si256((__m256i*)int_data1_64); // Невыровненные данные
__m256i vec_int64_2 = _mm256_load_si256((__m256i*)int_data2_64); // Выровненные данные
// 64-битные числа с плавающей точкой (двойная точность)
__m256d vec_double1 = _mm256_loadu_pd(double_data1); // Невыровненные данные
__m256d vec_double2 = _mm256_load_pd(double_data2);  // Выровненные данные
```



Выравнивание – это важно



Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0	
int a[1]				int a[0]				0x200000000
				int a[2]				0x200000008
								0x200000010
								0x200000018
short int b[3]	short int b[2]	short int b[1]	short int b[0]					0x200000020
		short int b[6]	short int b[5]	short int b[4]				0x200000028

32-byte aligned memory

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0	
int a[1]				int a[0]				0x200000000
short int b[1]	short int b[0]	int a[2]						0x200000008
short int b[5]	short int b[4]	short int b[3]	short int b[2]					0x200000010
long int c[0] ...						short int b[6]		0x200000018
						long int c[0]...		0x200000020
								0x200000028

unaligned memory

Представление памяти

Пример из документации Intel:

```
extern __m256i _mm256_load_si256(__m256i const *a);
```

Arguments

***a**

pointer to a memory location that can hold constant integer values; the address must be 32-byte aligned



Нумерация в векторе

`__m512i vector=_mm512_set_epi64`

`(Vec[7],Vec[6],Vec[5],Vec[4],Vec[3],Vec[2],Vec[1],Vec[0])`

511	256	255	128	127	0
ZMM0	YMM0	XMM0			
ZMM1	YMM1	XMM1			
ZMM2	YMM2	XMM2			

Представление регистра

xmm0 1 2

xmm1 6 2



xmm2 7 4

`xmm2 = xmm0+xmm1`

xmm register
SSE2

ymm0 1 2 3 4

ymm1 6 2 1 5



ymm2 7 4 4 9

`ymm2 = ymm0+ymm1`

ymm register
AVX2

zmm0 1 2 3 4 5 6 7 8

zmm1 6 2 2 3 4 2 1 1



zmm2 7 4 2 7 9 8 8 9

`zmm2 = zmm0+zmm1`

zmm register
AVX512

Вывод вектора



Исходный код

```
// Указание компилятору использовать инструкции AVX2
#pragma GCC target("avx2")

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

// Функция для вывода содержимого AVX2 регистра (__m256i) в виде
// массива 32-разрядных целых чисел
void print_avx2_register(const char *pname, __m256i r) {
    int vals[8] __attribute__((aligned(32)));

    // Сохранение содержимого регистра в массив
    mm256_store_si256((__m256i*)vals, r);

    // Вывод содержимого массива
    printf("Содержимое регистра %s:\n", pname);
    for (int i = 0; i < 8; i++) {
        printf("%s[%d] = %d\n", pname, i, vals[i]);
    }
}

int main() {
    // Используем mm256_set_epi32 для задания значений в векторе
    __m256i vec = mm256_set_epi32(8, 7, 6, 5, 4, 3, 2, 1);

    // Вывод содержимого регистра
    print_avx2_register("vec", vec);

    return 0;
}
```

Выполнение программы

```
$ gcc -O2 -march=native -o print_vec 01_avx2_set_print.c
$ ./print_vec
Содержимое регистра vec:
vec[0] = 1
vec[1] = 2
vec[2] = 3
vec[3] = 4
vec[4] = 5
vec[5] = 6
vec[6] = 7
vec[7] = 8
$
```



Простые операции



- `_mm256_storeu_si256(dest, src)` Интринсик для сохранения 256-битного вектора данных из регистра в память. (dest может быть невыровненным адресом)
- `_mm256_add_epi32 (src1, src2)` Интринсик для сложения 32-битных целых чисел в AVX2-регистре.
- `_mm256_max_epi32(src1, src2)` Поиск максимума среди 32-битных целых чисел в 2-х AVX2-регистрах.
- `_mm256_set1_epi32(int)` Задаёт всему вектору представление 32 битными значениями int

В четвертом примере мы воспользуемся ещё и этими интринсиками:

- `_mm256_cmpeq_epi32(src1, src2)` Сравнивает соответствующие элементы двух 256-битных векторов src1 и src2.
- `_mm256_castsi256_ps(src1)` Преобразует вектор из целочисленного формата (`__m256i`) в формат с плавающей точкой (`__m256`).
- `_mm256_movemask_ps (src1)` Эта функция извлекает знаковые биты (старшие биты) каждого 32-битного элемента вектора.



Поиск максимального значения



Бенчмарк

```
// Универсальная функция для бенчмаркинга
double benchmark(int (*func)(const int*, int), int* array, int size, int repetitions) {
    double acc = 0; // Переменная для накопления времени исполнения относительно time.h
    uint64_t acct = 0; // Переменная для замера среднего количества тактов (TSC) на одну итерацию
    uint64_t startt = 0; // Переменная для значения (TSC) до вызова функции
    uint64_t endt = 0; // Переменная для значения (TSC) после вызова функции
    for (int i = 0; i < repetitions; ++i) {
        initialize_array(array, size); // Чтобы на каждой итерации работать с новым массивом
        uint64_t start_time = start_timer(); // Старт высокоточного замера времени
        startt = __rdtsc(); // Начало замера тактов
        func(array, size); // Вызов функции
        endt = __rdtsc(); // Окончание замера тактов
        uint64_t elapsed_time_ns = end_timer(start_time); // Окончание замера времени
        acct += endt - startt; // Накопление тактов за итерацию
        acc += (double)elapsed_time_ns / 1000000000.0; // Накопление времени в секундах
    }
    printf("Tacts for one repetition: %lu\n", acct / repetitions);
    return acc;
}
```

Функция Main()

Подготовка массива данных

```
// Функция для инициализации массива случайными числами
void initialize_array(int* array, int size) {
    for (int i = 0; i < size; ++i) {
        array[i] = rand() % 10000; // Случайные числа от 0 до 9999
    }
}
```

```
int main() {
    int data[SIZE]; // Статический массив
    printf("Measuring %d reps of finds max in %d-sized array\n", REPT, SIZE);
    srand(time(NULL));
    initialize_array(data, SIZE);
    // Бенчмарк для SISD
    double time_sisd = benchmark(find_max_sisd, data, SIZE, REPT);
    printf("Measured (SISD): %f seconds\n\n", time_sisd);
    // Бенчмарк для SIMD
    double time_simd = benchmark(find_max_simd, data, SIZE, REPT);
    printf("Measured (SIMD): %f seconds\n\n", time_simd);
    if (time_simd > 0) {
        printf("SIMD speedup against SISD = %.2f \n\n", time_sisd / time_simd);
    } else {
        printf("SIMD time is 0!\n\n");
    }
    return 0;
}
```



Поиск максимального значения SISD

```
// Функция для поиска максимума с использованием SISD (обычный подход)
int find_max_sisd(const int* array, int size) {
    int max_value = array[0];
    for (int i = 1; i < size; i++) {
        if (array[i] > max_value) {
            max_value = array[i];
        }
    }
    return max_value;
}
```

Исходный код

**Дизассемблированный
код**

SISD-реализация: 9 строк кода. 29 строк ассемблера

```
find_max_sisd:
    movl    (%rcx), %eax
    cmpl    $1, %edx
    jle     .L1
    subl    $2, %edx
    leaq    4(%rcx), %r8
    leaq    8(%rcx,%rdx,4), %r9
    movq    %r9, %rdx
    subq    %r8, %rdx
    andl    $4, %edx
    je      .L3
    movl    (%r8), %edx
    leaq    8(%rcx), %r8
    cmpl    %edx, %eax
    cmovl    %edx, %eax
    cmpq    %r9, %r8
    je      .L1
.L3:
    movl    (%r8), %edx
    cmpl    %edx, %eax
    cmovl    %edx, %eax
    movl    4(%r8), %edx
    cmpl    %edx, %eax
    cmovl    %edx, %eax
    addq    $8, %r8
    cmpq    %r9, %r8
    jne     .L3
.L1:
    ret
```

Поиск максимального значения SIMD



```
// Функция для поиска максимума с использованием AVX2 (SIMD)
int find_max_simd(const int* array, int size) {
    __m256i max_vec = _mm256_set1_epi32(array[0]);

    int i = 0;
    for (; i <= size - 8; i += 8) {
        __m256i vec = _mm256_loadu_si256((__m256i*)&array[i]);
        max_vec = _mm256_max_epi32(max_vec, vec);
    }
    // Мы обработали кратную 8 часть массива получили
    // 8 максимальных значений, сравним кто из них больше.
    int max_vals[8];
    _mm256_storeu_si256((__m256i*)max_vals, max_vec);

    int max_value = max_vals[0];
    for (int j = 1; j < 8; j++) {
        if (max_vals[j] > max_value) {
            max_value = max_vals[j];
        }
    }
    // обрабатываем оставшиеся элементы сравним с «векторным
    // максимумом»
    for (; i < size; i++) {
        if (array[i] > max_value) {
            max_value = array[i];
        }
    }
    return max_value;
}
```

Исходный код

SIMD реализация: 26 строк кода,
74 строки ассемблера

```
find_max_simd:
    subq    $40, %rsp
    movl    (%rcx), %eax
    addq    $8, %rdx
    movq    %rcx, %r9
    movl    %edx, %r8d
    vmovd    %eax, %xmm1
    vpbroadcastb    %xmm1, %ymm1
    cmpl    $7, %edx
    jle     .L17
    movq    %rcx, %rax
    leal    -8(%rdx), %ecx
    shrl    $3, %ecx
    movl    %ecx, %edx
    salq    $5, %rdx
    leaq    32(%r9,%rdx), %rdx
.L13:
    vpmasxd (%rax), %ymm1, %ymm0
    addq    $32, %rax
    vmovdqa %ymm0, %ymm1
    cmpq    %rdx, %rax
    jne     .L13
    leal    8(%rcx,8), %r11d
    vmovd    %xmm0, %eax
.L12:
    vmovdqu %ymm1, (%rsp)
    leaq    32(%rsp), %r10
    movl    4(%rsp), %edx
    cmpl    %edx, %eax
    cmovl    %edx, %eax
    leaq    8(%rsp), %rdx
.L14:
    movl    (%rdx), %ecx
    cmpl    %ecx, %eax
    cmovl    %ecx, %eax
    movl    4(%rdx), %ecx
    cmpl    %ecx, %eax
    cmovl    %ecx, %eax
    addq    $8, %rdx
    cmpq    %r8, %rdx
    jne     .L16
.L11:
    vzeroupper
    addq    $40, %rsp
    ret
.L17:
    xorl    %r11d, %r11d
    jmp     .L12
```



Сравнение производительности



```
$ gcc -O2 -march=native -o max_bench 03_max_bench.c
$ ./max_bench
Measuring 10000 reps of finds max in 65536-sized array
Tacts for one repetition: 67821
Measured (SISD): 0.321435 seconds

Tacts for one repetition: 5851
Measured (SIMD): 0.027934 seconds

SIMD speedup against SISD = 11.51
```

Несмотря на более сложную в написании и по объёму функцию,
код выполняется в ~10 раз быстрее.



Векторизуем поиск

Классическая реализация

```
int find_sisd(const int* array, int size, int key) {
    int i;
    for (i = 0; i < size; i++)
        if (array[i] == key)
            return i;
    return -1;
}
```

Бенчмарк

```
double benchmark(int (*func)(const int*, int, int), const int* array, int size, int
repetitions) {
    double acc = 0; // Переменная для накопления времени исполнения относительно time.h
    uint64_t acct = 0; // Замер среднего количества тактов (TSC) на одну итерацию
    uint64_t startt = 0; // Переменная для значения (TSC) до вызова функции
    uint64_t endt = 0; // Переменная для значения (TSC) после вызова функции
    volatile int key = 0;
    volatile int result = 0;
    for (int i = 0; i < repetitions; ++i) {
        key = rand() % SIZE; // Выбираем новый ключ
        clock_t start = clock(); // Начало замера времени
        startt = _rdtsc(); // Начало замера тактов
        result = func(array, size, key); // Вызов функции
        endt = _rdtsc(); // Окончание замера тактов
    }
    ...
}
```

Заполнение массива

```
void initialize_array(int* array, int size) {
    for (int i = 0; i < size; ++i) {
        array[i] = i;
    }
}
```

Векторная реализация



```
static inline int first_equal_yvalue(__m256i src1, __m256i src2) {
    __m256i cmp_result = _mm256_cmpeq_epi32(src1, src2);
    int mask = _mm256_movemask_ps(_mm256_castsi256_ps(cmp_result));
    if (mask != 0) {
        return __builtin_ctz(mask); // позиция первого ненулевого
    }
    return -1;
}

int find_simd(const int* array, int size, int key) {
    __m256i vec_i = _mm256_set1_epi32(key);
    int j = 0; // Для обработки векторизуемой части массива
    // векторная часть
    for (int j = 0; j < size - 8; j += 8) {
        __m256i vec_a = _mm256_loadu_si256((__m256i*)&array[j]);
        int pos = first_equal_yvalue(vec_a, vec_i);
        if (pos != -1) {
            return j + pos;
        }
    }
    // Обработка "хвостика"
    for (int j = limit; j < size; ++j) {
        if (array[j] == key) {
            return j;
        }
    }
    return -1;
}
```



Работа с маской

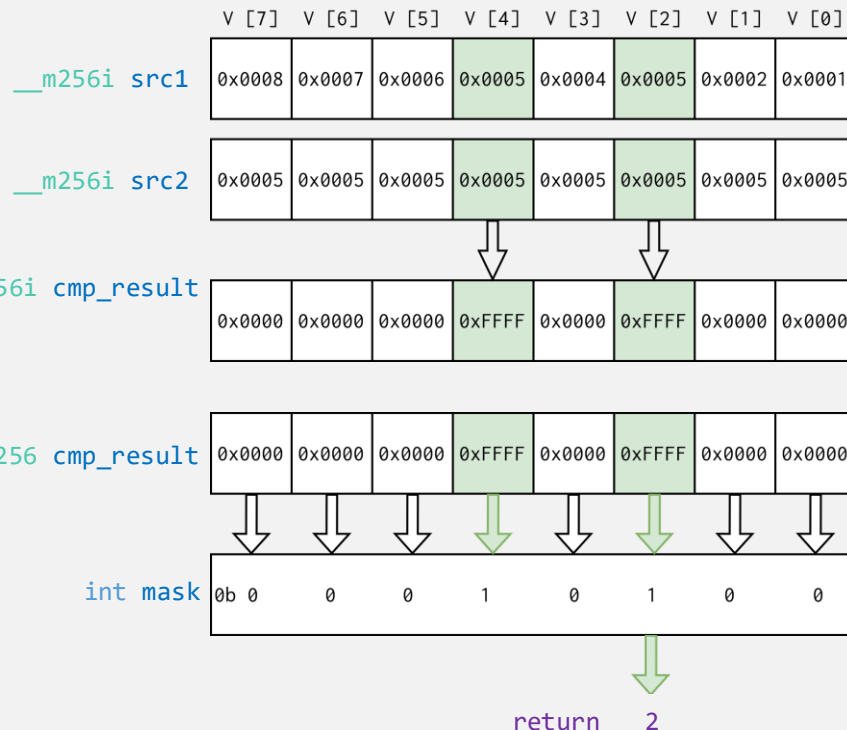


```
static inline int first_equal_yvalue(__m256i src1, __m256i src2) {  
    __m256i cmp_result = _mm256_cmpeq_epi32(src1, src2);
```

```
    int mask = _mm256_movemask_ps(_mm256_castsi256_ps(cmp_result));
```

```
    _mm256_castsi256_ps(cmp_result) -> __m256 cmp_result
```

```
    if (mask != 0) {  
        return __builtin_ctz(mask);    }  
    return -1;  
}
```



Результаты Find



```
$ gcc -O2 -march=native -o find_bench 04_find_bench.c
$ ./find_bench
Measuring 10000 reps of finds key in 65536-sized array
key= 25819,find= 25819,
Tacts for one repetition: 16964
Measured (SISD): 0.080535 seconds

key= 14265,find= 14265,
Tacts for one repetition: 4121
Measured (SIMD): 0.019709 seconds

SIMD speedup against SISD = 4.09
```



Оптимизации компилятора -O3 -mavx2



Добавим -O3:

```
$ gcc -O3 -march=native -o find_bench 04_find_bench.c
$ ./find_bench
Measuring 10000 reps of finds key in 65536-sized array
key= 41953,find= 41953,
Tacts for one repetition: 16775
Measured (SISD): 0.079647 seconds

key= 39285,find= 39285,
Tacts for one repetition: 3332
Measured (SIMD): 0.015987 seconds

SIMD speedup against SISD = 4.98
```

Добавим -funroll-loops:

```
$ gcc -O3 -march=native -funroll-loops -o find_bench 04_find_bench.c
$ ./find_bench
Measuring 10000 reps of finds key in 65536-sized array
key= 16100,find= 16100,
Tacts for one repetition: 11130
Measured (SISD): 0.052956 seconds

key= 18180,find= 18180,
Tacts for one repetition: 2560
Measured (SIMD): 0.012332 seconds

SIMD speedup against SISD = 4.29
```



Где же наш % прироста производительности?



- Архитектурные особенности процессоров
- Использование «устаревшей» технологии.
Отсутствие в AVX2 набора интринсиков для прямой работы с маской
- Работа с выровненными данными
- Вектор на 8 элементов (в максимальном случае вектор `__mm512i` вмещает 64 элемента типа `char`)



А что уже есть в PostgreSQL?



REL_16_STABLE → REL_17_STABLE

```
git log --pretty=format:"%h - %an, %ar : %s" --  
src/include/port/simd.h --relative-date
```

- 29275b1d17 - Bruce Momjian, 8 месяцев назад : Update copyright for 2024
- ef7002dbe0 - Michael Paquier, 1 год, 7 месяцев назад : Fix various typos in code and tests
- c8e1ba736b - Bruce Momjian, 1 год, 8 месяцев назад : Update copyright for 2023
- 73b9d051c6 - John Naylor, 2 года назад : Fix sign-compare warnings arising from port/simd.h
- 865424627d - John Naylor, 2 года назад : Further code review of port/simd.h
- c6a43c25a8 - John Naylor, 2 года назад : Fix broken cast on MSVC
- 82739d4a80 - John Naylor, 2 года назад : Use ARM Advanced SIMD (NEON) intrinsics where available
- f8f19f7086 - John Naylor, 2 года назад : Abstract some more architecture-specific details away from SIMD functionality
- 121d2d3d70 - John Naylor, 2 года, 1 месяц назад : Use SSE2 in is_valid_ascii() where available.
- e813e0e168 - John Naylor, 2 года, 1 месяц назад : Add optimized functions for linear search within byte arrays
- 56f2c7b58b - John Naylor, 2 года, 1 месяц назад : Support SSE2 intrinsics where available

- de7c6fe834 - John Naylor, 6 месяцев назад : Fix signedness error in 9f225e992 for gcc
- 9f225e992b - John Naylor, 6 месяцев назад : Introduce helper SIMD functions for small byte arrays

коммиты из REL_16_STABLE

- 29275b1d17 - Bruce Momjian, 8 месяцев назад : Update copyright for 2024
- ef7002dbe0 - Michael Paquier, 1 год, 7 месяцев назад : Fix various typos in code and tests
- c8e1ba736b - Bruce Momjian, 1 год, 8 месяцев назад : Update copyright for 2023
- 73b9d051c6 - John Naylor, 2 года назад : Fix sign-compare warnings arising from port/simd.h
- 865424627d - John Naylor, 2 года назад : Further code review of port/simd.h
- c6a43c25a8 - John Naylor, 2 года назад : Fix broken cast on MSVC
- 82739d4a80 - John Naylor, 2 года назад : Use ARM Advanced SIMD (NEON) intrinsics where available
- f8f19f7086 - John Naylor, 2 года назад : Abstract some more architecture-specific details away from SIMD functionality
- 121d2d3d70 - John Naylor, 2 года, 1 месяц назад : Use SSE2 in is_valid_ascii() where available.
- e813e0e168 - John Naylor, 2 года, 1 месяц назад : Add optimized functions for linear search within byte arrays
- 56f2c7b58b - John Naylor, 2 года, 1 месяц назад : Support SSE2 intrinsics where available



И для чего используется simd.h в Postgres?



PostgreSQL 16:

- pg_lfind.h
 - pg_lfind8
 - pg_lfind8_le
 - pg_lfind32
- ascii.h
 - is_valid_ascii()

PostgreSQL 17:

- pg_lfind.h
 - pg_lfind8
 - pg_lfind8_le
 - pg_lfind32
- ascii.h
 - is_valid_ascii()
- radixtree.h
 - RT_NODE_16_SEARCH_EQ()
 - T_NODE_16_GET_INSERTPOS()



Что можно добавить?



Агрегатные функции

- Суммирование (SUM)
- Среднее (AVG)
- Минимум/максимум (MIN/MAX)

Поиск и фильтрация данных

- Операции сравнения
- LIKE и ILIKE

Манипуляции со строками

- Преобразования регистров
- Конкатенация строк

Алгоритмы сортировки

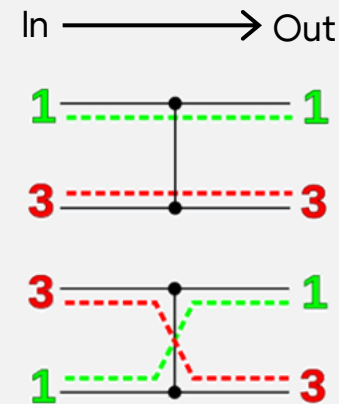
- Сортировка (ORDER BY)

Обработка JSON и JSONB

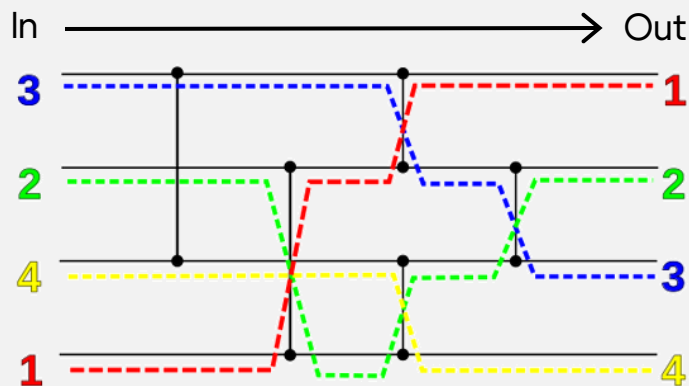
- Извлечение данных



Сортирующие сети



Компаратор

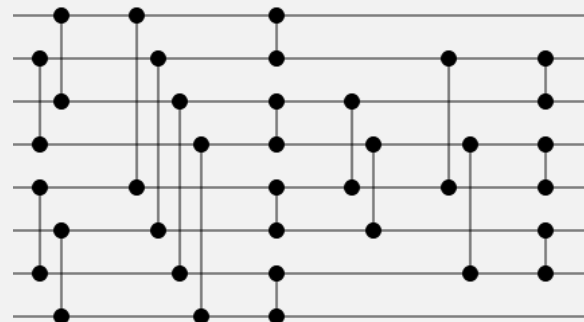


$[(0,2),(1,3)]$

$[(0,1),(2,3)]$

$[(1,2)]$

Оптимальная сеть – 4 элемента



$[(0,2),(1,3),(4,6),(5,7)]$

$[(0,4),(1,5),(2,6),(3,7)]$

$[(0,1),(2,3),(4,5),(6,7)]$

$[(2,4),(3,5)]$

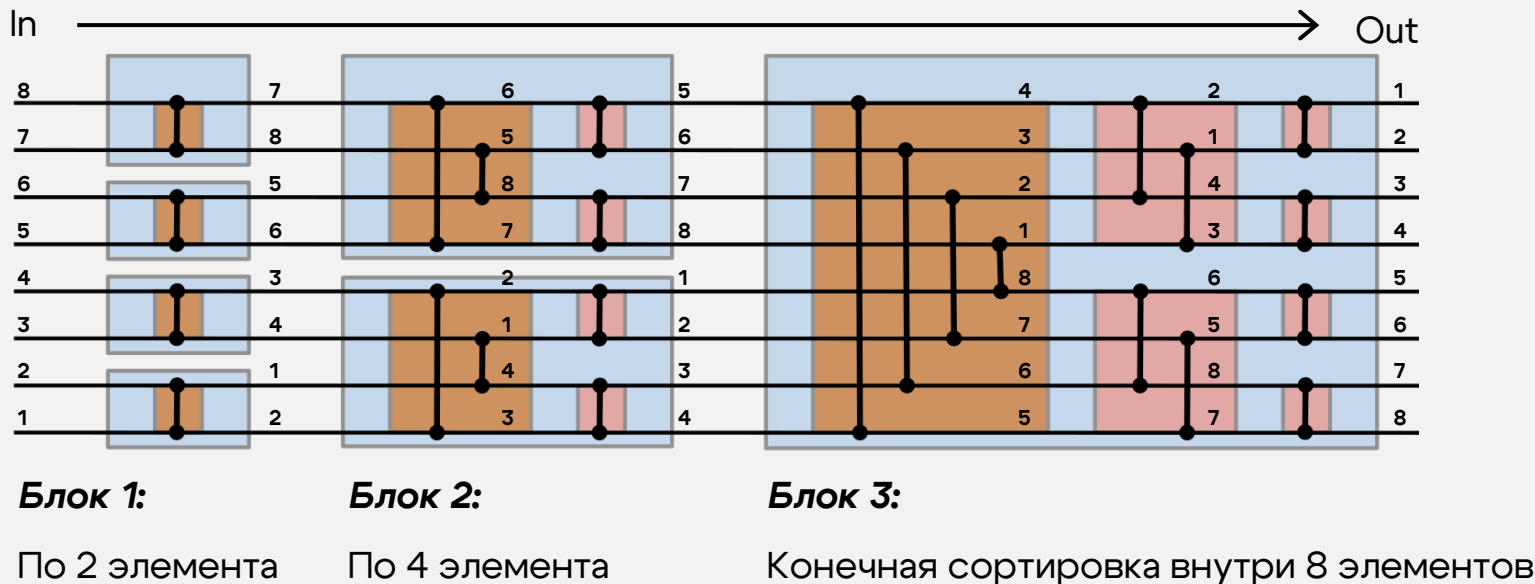
$[(1,4),(3,6)]$

$[(1,2),(3,4),(5,6)]$

Оптимальная сеть – 8 элементов

Битонная сортировка (Bitonic sort)

Разработана американским информатиком Кеннетом Бэтчером в 1968 году.
 $O(\log^2 n)$, где n — число элементов для сортировки.



Векторная сортировка (перестановки)

Визуальное представление сети

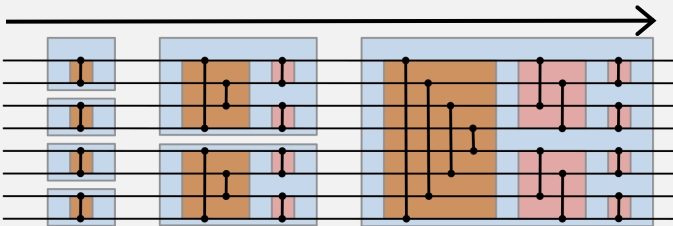


Схема соединений

Шаг 1: [(0,1),(2,3),(4,5),(6,7)]

Шаг 2: [(0,3),(1,2),(4,7),(5,6)]

Шаг 3: [(0,1),(2,3),(4,5),(6,7)]

Шаг 4: [(0,7),(1,6),(2,5),(3,4)]

Шаг 5: [(0,2),(1,3),(4,6),(5,7)]

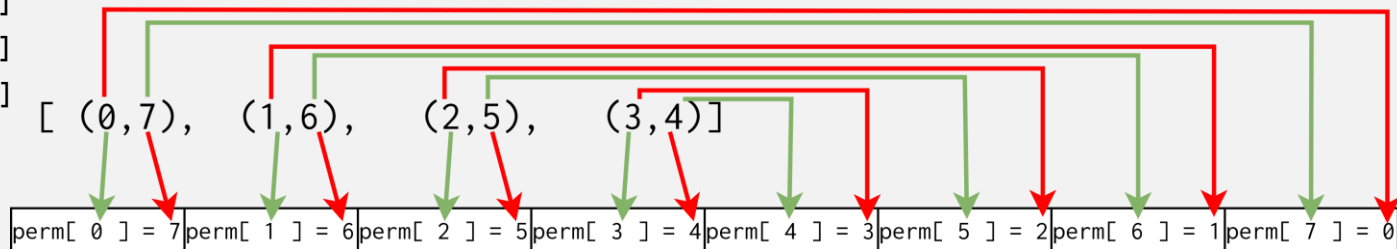
Шаг 6: [(0,1),(2,3),(4,5),(6,7)]

Итоговые массивы перестановок

```
// Выровненные константы для перестановок
ALIGNED_32 int permute_step_1_1[8] = {1, 0, 3, 2, 5, 4, 7, 6};
ALIGNED_32 int permute_step_2_1[8] = {3, 2, 1, 0, 7, 6, 5, 4};
ALIGNED_32 int permute_step_2_2[8] = {1, 0, 3, 2, 5, 4, 7, 6};
ALIGNED_32 int permute_step_3_1[8] = {7, 6, 5, 4, 3, 2, 1, 0};
ALIGNED_32 int permute_step_3_2[8] = {2, 3, 0, 1, 6, 7, 4, 5};
ALIGNED_32 int permute_step_3_3[8] = {1, 0, 3, 2, 5, 4, 7, 6};
```

Составление массива перестановок на примере

Шага 4, в коде `permute_step_3_1`: Шаг 4: [(0,7),(1,6),(2,5),(3,4)]



Векторная сортировка (Маски смещения)

Визуальное представление сети

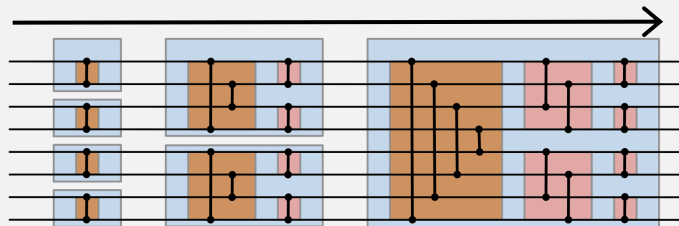


Схема соединений

Шаг 1: [(0,1),(2,3),(4,5),(6,7)]

Шаг 2: [(0,3),(1,2),(4,7),(5,6)]

Шаг 3: [(0,1),(2,3),(4,5),(6,7)]

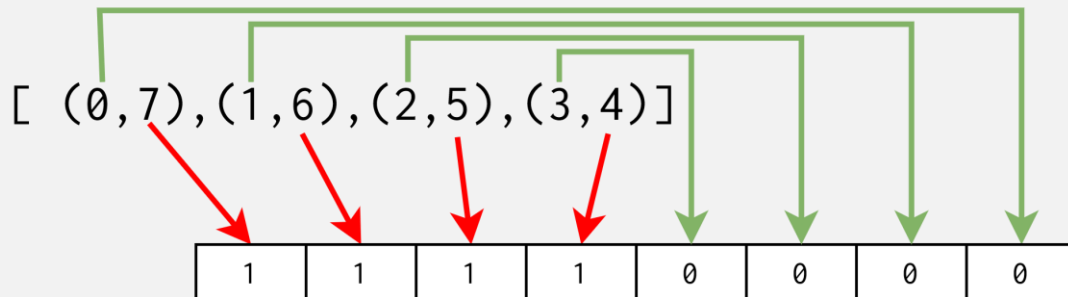
Шаг 4: [(0,7),(1,6),(2,5),(3,4)]

Шаг 5: [(0,2),(1,3),(4,6),(5,7)]

Шаг 6: [(0,1),(2,3),(4,5),(6,7)]

Составление маски смещения на примере

Шаг 4: Шаг 4: [(0,7),(1,6),(2,5),(3,4)]



Blend_mask_step4 =

0b11110000

Векторная сортировка (компаратор)



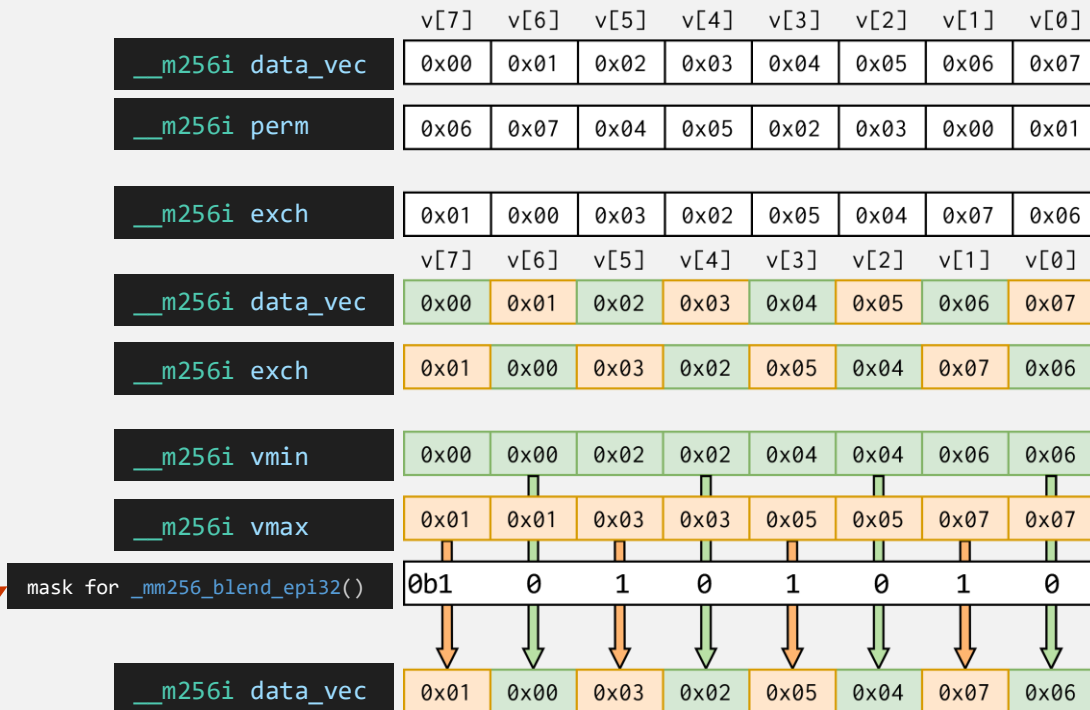
Код компаратора для первого шага

```
__m256i perm =  
_mm256_load_si256((__m256i*)permute_step_1_1);
```

```
__m256i exch =  
_mm256_permutevar8x32_epi32(data_vec, perm);
```

```
__m256i vmin = _mm256_min_epi32(data_vec, exch);  
__m256i vmax = _mm256_max_epi32(data_vec, exch);
```

```
data_vec =  
_mm256_blend_epi32(vmin, vmax, 0b10101010);
```



Векторная сортировка (основная функция)



Код векторной части сортировки

```
// Функция для сортировки блоков по 8 элементов с использованием
битонической сортировки
void bitonic_sort_8(int* data) {
    // Используем выровненные команды для загрузки данных
    __m256i data_vec = _mm256_load_si256((__m256i*)data); //
    Выравненная загрузка

    // Шаг 1: перестановка и минимизация/максимизация
    __m256i perm = _mm256_load_si256((__m256i*)permute_step_1_1);
    __m256i exch = _mm256_permutevar8x32_epi32(data_vec, perm);
    __m256i vmin = _mm256_min_epi32(data_vec, exch);
    __m256i vmax = _mm256_max_epi32(data_vec, exch);
    data_vec = _mm256_blend_epi32(vmin, vmax, 0b10101010);

    // Шаг 2
    perm = _mm256_load_si256((__m256i*)permute_step_2_1);
    exch = _mm256_permutevar8x32_epi32(data_vec, perm);
    vmin = _mm256_min_epi32(data_vec, exch);
    vmax = _mm256_max_epi32(data_vec, exch);
    data_vec = _mm256_blend_epi32(vmin, vmax, 0b11001100);

    perm = _mm256_load_si256((__m256i*)permute_step_2_2);
    exch = _mm256_permutevar8x32_epi32(data_vec, perm);
    vmin = _mm256_min_epi32(data_vec, exch);
    vmax = _mm256_max_epi32(data_vec, exch);
    data_vec = _mm256_blend_epi32(vmin, vmax, 0b10101010);

    // Шаг 3
    perm = _mm256_load_si256((__m256i*)permute_step_3_1);
    exch = _mm256_permutevar8x32_epi32(data_vec, perm);
    vmin = _mm256_min_epi32(data_vec, exch);
    vmax = _mm256_max_epi32(data_vec, exch);
    data_vec = _mm256_blend_epi32(vmin, vmax, 0b11110000);

    perm = _mm256_load_si256((__m256i*)permute_step_3_2);
    exch = _mm256_permutevar8x32_epi32(data_vec, perm);
    vmin = _mm256_min_epi32(data_vec, exch);
    vmax = _mm256_max_epi32(data_vec, exch);
    data_vec = _mm256_blend_epi32(vmin, vmax, 0b11001100);

    perm = _mm256_load_si256((__m256i*)permute_step_3_3);
    exch = _mm256_permutevar8x32_epi32(data_vec, perm);
    vmin = _mm256_min_epi32(data_vec, exch);
    vmax = _mm256_max_epi32(data_vec, exch);
    data_vec = _mm256_blend_epi32(vmin, vmax, 0b10101010);

    // Сохранение отсортированных данных
    _mm256_store_si256((__m256i*)data, data_vec);
}
```



Сравнение производительности



```
$ gcc -O2 -march=native -o net_sort 05_net_sort_bench.c
$ ./net_sort
Measuring 1000000 reps of finds in 8-sized array
tacts: 152
Measured (SISD): 0.204396 seconds

SISD SORTED.
tacts: 15
Measured (SIMD): 0.138900 seconds

SIMD SORTED.
SIMD speedup against SISD = 1.47
```

Здесь мы можем наблюдать неточность таймера на коротких замерах.

Я бы больше верил процессорному счётчику, согласно которому ускорение примерно в 10 раз.



Сравнение производительности

(дополнительный пример)



```
$ gcc -O2 -march=native -o net_sort_adv 06_net_sort_bench_advanced.c
$ ./net_sort_adv
Measuring 10000 reps of finds in 4096-sized array
tacts: 430216
Measured (SISD): 2.037263 seconds
SISD SORTED.

tacts: 361005
Measured (SISD_quicksort): 1.709801 seconds
SISD QUICK SORTED.

tacts: 302280
Measured (SIMD): 1.431510 seconds
SIMD SORTED.

SIMD speedup against SISD = 1.42
SIMD speedup against QUICKSORT SISD = 1.19
```



Два мира



Google Highway library



Peloton DB

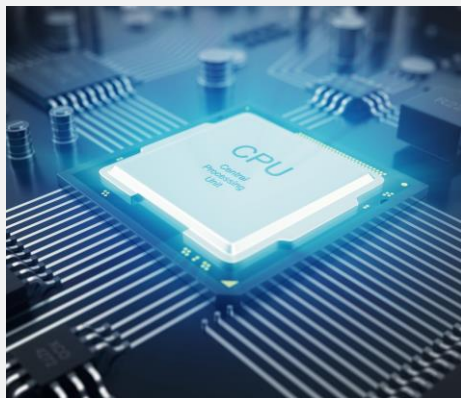


PG-Strom

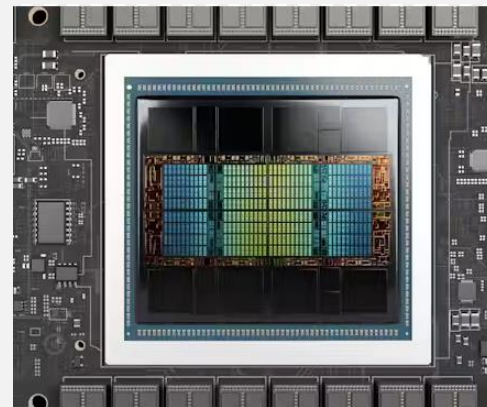


オモシロ技術を、カタチにする。

Brytlyt



CPU



GPU



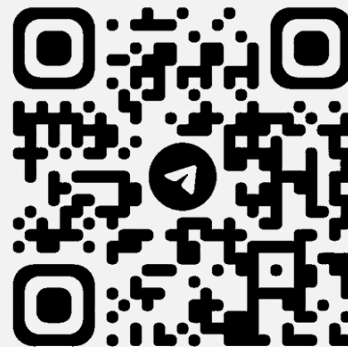
PG BootCamp Russia 2024 Kazan



PGBootCamp.ru

Спасибо!

Артем Бугаенко, «Тантор Лабс»



@BUGGAI