

Лекция 14

Векторизация кода

Intel SSE/AVX

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com

WWW: www.mkurnosov.net

Курс «Параллельные вычислительные технологии»

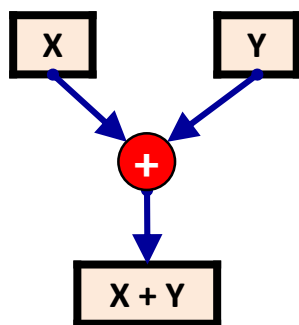
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Весенний семестр, 2017

Векторные процессоры

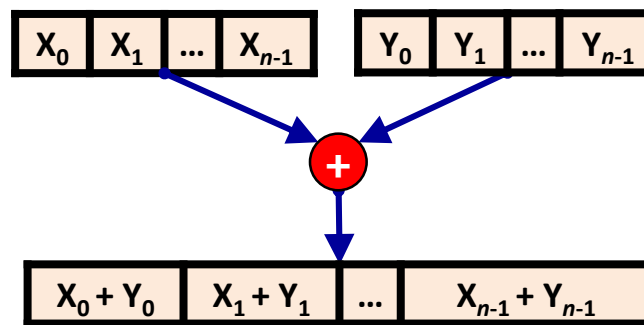
- **Векторный процессор (vector processor)** – процессор, поддерживающий на уровне системы команд операции для работы с одномерными массивами (векторами)

Скалярный процессор
(Scalar processor)



`add Z, X, Y`

Векторный процессор
(Vector processor)



`add.v Z[0:n-1], X[0:n-1], Y[0:n-1]`

Векторный процессор vs. Скалярный процессор

Поэлементное суммирование двух массивов из 10 чисел

Скалярный процессор (scalar processor)

```
for i = 1 to 10 do
  IF - Instruction Fetch (next)
  ID - Instruction Decode
  Load Operand1
  Load Operand2
  Add Operand1 Operand2
  Store Result
end for
```

Векторный процессор (vector processor)

```
IF - Instruction Fetch
ID - Instruction Decode
Load Operand1[0:9]
Load Operand2[0:9]
Add Operand1[0:9] Operand2[0:9]
Store Result
```

- Меньше преобразований адресов
- Меньше IF, ID
- Меньше конфликтов конвейера, ошибок предсказания переходов
- Эффективнее доступ к памяти (2 выборки vs. 20)
- Операция над операндами выполняется параллельно
- Уменьшился размер кода

Производительность векторных процессоров

Факторы влияющие на производительность векторного процессора

- Доля кода в векторной форме
- Длина вектора (векторного регистра)
- Латентность векторной инструкции (vector startup latency) – начальная задержка конвейера при обработке векторной инструкции
- Количество векторных регистров
- Количество векторных модулей доступа к памяти (load-store)
- ...

Классификация векторных систем

- **Векторные процессоры память-память**
(memory-memory vector processor) – векторы размещены в оперативной памяти, все векторные операции память-память
- Примеры:
 - ❑ CDC STAR-100 (1972, вектор 65535 элементов)
 - ❑ Texas Instruments ASC (1973)
- **Векторные процессоры регистр-регистр**
(register-vector vector processor) – векторы размещены в векторных регистрах, все векторные операции выполняются между векторными регистрами
- Примеры: практически все векторные системы начиная с конца 1980-х: Cray, Convex, Fujitsu, Hitachi, NEC, ...

Векторные вычислительные системы

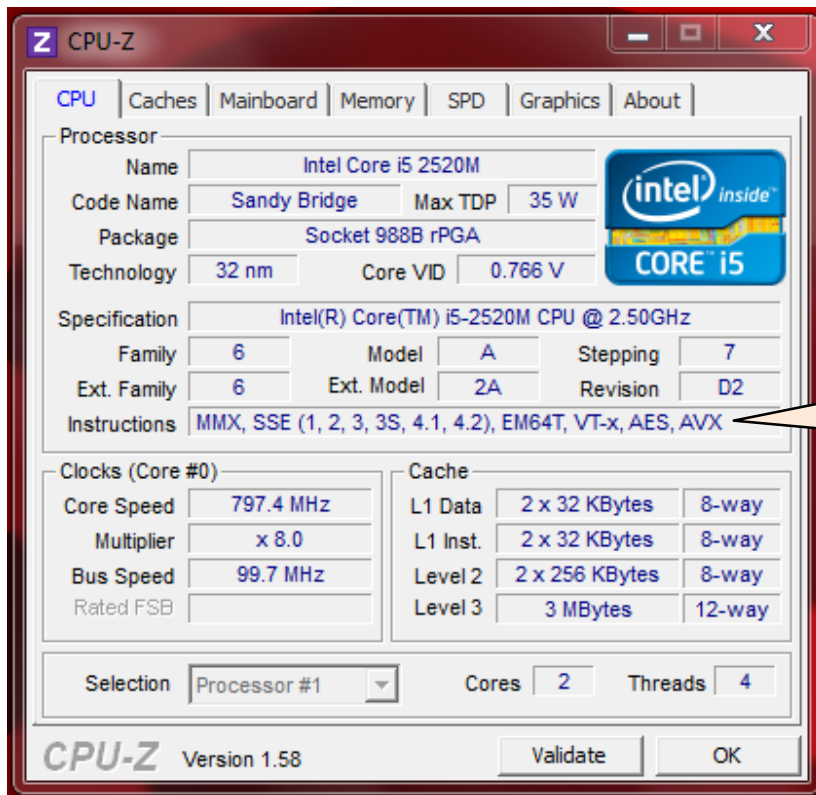
- Cray 1 (1976) 80 MHz, 8 regs, 64 elems
- Cray XMP (1983) 120 MHz 8 regs, 64 elems
- Cray YMP (1988) 166 MHz 8 regs, 64 elems
- Cray C-90 (1991) 240 MHz 8 regs, 128 elems
- Cray T-90 (1996) 455 MHz 8 regs, 128 elems
- Conv. C-1 (1984) 10 MHz 8 regs, 128 elems
- Conv. C-4 (1994) 133 MHz 16 regs, 128 elems
- Fuj. VP200 (1982) 133 MHz 8-256 regs, 32-1024 elems
- Fuj. VP300 (1996) 100 MHz 8-256 regs, 32-1024 elems
- NEC SX/2 (1984) 160 MHz 8+8K regs, 256+var elems
- NEC SX/3 (1995) 400 MHz 8+8K regs, 256+var elems

SIMD-инструкции современных процессоров

- Intel **MMX**: 1997, Intel Pentium MMX, IA-32
- AMD **3DNow!**: 1998, AMD K6-2, IA-32
- Apple, IBM, Motorola **AltiVec**: 1998, PowerPC G4, G5, IBM Cell/POWER
- Intel **SSE** (Streaming SIMD Extensions): 1999, Intel Pentium III
- Intel **SSE2**: 2001, Intel Pentium 4, IA-32
- Intel **SSE3**: 2004, Intel Pentium 4 Prescott, IA-32
- Intel **SSE4**: 2006, Intel Core, AMD K10, x86-64
- AMD **SSE5** (XOP, FMA4, CVT16): 2007, 2009, AMD Bulldozer
- Intel **AVX**: 2008, Intel Sandy Bridge
- ARM **Advanced SIMD (NEON)**: ARMv7, ARM Cortex A
- MIPS **SIMD Architecture (MSA)**: 2012, MIPS R5
- Intel **AVX2**: 2013, Intel Haswell
- Intel **AVX-512**: 2013, Intel Xeon Skylake, Intel Xeon Phi
- **ARMv8 -- Scalable Vector Extension (SVE, 2016)**

CPUID (CPU Identification): Microsoft Windows

Windows CPU-Z



- MMX
- SSE
- AVX
- AES
- ...

CPUID (CPU Identification): GNU/Linux

- Файл `/proc/cpuinfo`: в поле `flags` хранится информация о процессоре
- Файл `/sys/devices/system/cpu/cpuX/microcode/processor_flags`
- Устройство `/dev/cpu/CPUNUM/cpuid`: чтение выполняется через `lseek` и `pread` (требуется загрузка модуля ядра `cpuid`)

```
$ cat /proc/cpuinfo
```

```
processor: 0
```

```
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
```

```
...
```

```
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov  
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm  
constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf  
eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm  
pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c rdrand  
lahf_lm ida arat epb pln pts dtherm tpr_shadow vnmi flexpriority ept vpid fsgsbase  
smep erms xsaveopt
```

CPUID (CPU Identification): GNU/Linux

```
inline void cpuid(int fn, unsigned int *eax, unsigned int *ebx,
                 unsigned int *ecx, unsigned int *edx)
{
    asm volatile("cpuid"
        : "=a" (*eax), "=b" (*ebx), "=c" (*ecx), "=d" (*edx)
        : "a" (fn));
}

int is_avx_supported()
{
    unsigned int eax, ebx, ecx, edx;

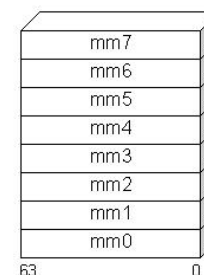
    cpuid(1, &eax, &ebx, &ecx, &edx);
    return (ecx & (1 << 28)) ? 1 : 0;
}

int main()
{
    printf("AVX supported: %d\n", is_avx_supported());
    return 0;
}
```

Intel 64 and IA-32 Architectures Software Developer's
Manual (Vol. 2A)

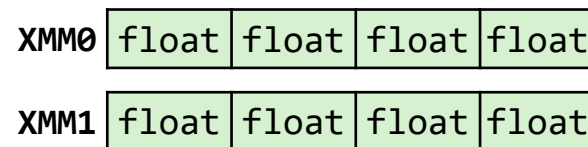
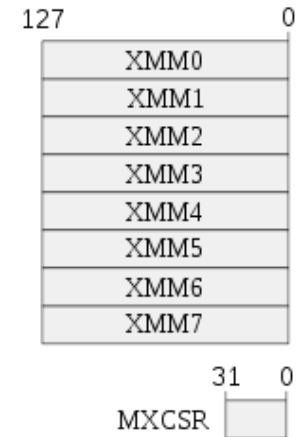
Intel MMX

- 1997, Intel Pentium MMX
- MMX – набор SIMD-инструкции для обработки целочисленных векторов длиной 64 бит
- 8 виртуальных регистров mm0, mm1, ..., mm7 – ссылки на физические регистры x87 FPU (ОС не требуется сохранять/восстанавливать регистры mm0, ..., mm7 при переключении контекста)
- Типы векторов: 8 x 1 char, 4 x short int, 2 x int
- MMX-инструкции разделяли x87 FPU с FP-инструкциями – требовалось оптимизировать поток инструкций (отдавать предпочтение инструкциям одного типа)



Intel SSE

- 1999, Pentium III
- 8 векторных регистров шириной 128 бит:
%xmm0, %xmm1, ..., %xmm7
- Типы данных: float (4 элемента на вектор)
- 70 инструкций: команды пересылки, арифметические команды, команды сравнения, преобразования типов, побитовые операции
- Инструкции явной предвыборки данных, контроля кэширования данных и контроля порядка операций сохранения



```
mulps %xmm1, %xmm0 // xmm0 = xmm0 * xmm1
```

Intel SSE

- Один из разработчиков расширения SSE – ***В.М. Пентковский (1946 – 2012 г.)***
- До переход в Intel являлся сотрудником Новосибирского филиала ИТМиВТ (программное обеспечение многопроцессорных комплексов Эльбрус 1 и 2, язык Эль-76, процессор Эль-90, ...)
- ❑ Jagannath Keshava and Vladimir Pentkovski: **Pentium III Processor Implementation Tradeoffs**. // Intel Technology Journal. — 1999. — Т. 3. — № 2.
- ❑ Srinivas K. Raman, Vladimir M. Pentkovski, Jagannath Keshava: **Implementing Streaming SIMD Extensions on the Pentium III Processor**. // IEEE Micro, Volume 20, Number 1, January/February 2000: 47-57 (2000)

Intel SSE2

- **2001, Pentium 4, IA32, x86-64 (Intel 64, 2004)**
- **16 векторных регистров шириной 128 бит:**
`%xmm0, %xmm1, ..., %xmm7; %xmm8, ..., %xmm15`
- Добавлено 144 инструкции к 70 инструкциям SSE
- По сравнению с SSE сопроцессор FPU (x87) обеспечивает более точный результат при работе с вещественными числами

16 x char	char	char	char	char	char	...	char
8 x short int	short int		short int		...		short int
4 x float int	float		float		float		float
2 x double	double				double		
1 x 128-bit int	128-bit integer						

Intel SSE3, SSE4

- **Intel SSE3: 2003, Pentium 4 Prescott, IA32, x86-64 (Intel 64, 2004)**
- Добавлено 13 новых инструкции к инструкциям SSE2
- Возможность горизонтальной работы с регистрами – команды сложения и вычитания нескольких значений, хранящихся в одном регистре
- **Intel SSE4: 2006, Intel Core, AMD Bulldozer**
- Добавлено 54 новых инструкции:
 - SSE 4.1: 47 инструкций, Intel Penryn
 - SSE 4.2: 7 инструкций, Intel Nehalem

Horizontal instruction

xmm0	a3	a2	a1	a0
------	----	----	----	----

xmm1	b3	b2	b1	b0
------	----	----	----	----

haddps %xmm1, %xmm0

xmm1	b3+b2	b1+b0	a3+a2	a1+a0
------	-------	-------	-------	-------

Intel AVX

- **2008, Intel Sandy Bridge (2011), AMD Bulldozer (2011)**
- Размер векторов увеличен до **256 бит**
- Векторные регистры переименованы: ymm0, ymm1, ..., ymm15
- Регистры xmm# – это младшие 128 бит регистров ymm#
- Трехоперандный синтаксис AVX-инструкций: $C = A + B$
- Использование ymm регистров требует поддержки со стороны операционной системы (для сохранения регистров при переключении контекстов)
 - Linux ядра $\geq 2.6.30$
 - Apple OS X 10.6.8
 - Windows 7 SP 1
- Поддержка компиляторами:
 - GCC 4.6
 - Intel C++ Compiler 11.1
 - Microsoft Visual Studio 2010
 - Open64 4.5.1

	255	128	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

Типы векторных инструкций Intel SSE/AVX

ADDPS



- **Название инструкции**

- **Тип инструкции**

S – над скаляром (scalar)

P – над упакованным вектором (packed)

- **ADDPS** – add 4 packed single-precision values (float)

- **ADDSD** – add 1 scalar double-precision value (double)

- **Тип элементов вектора/скаляра**

S – single precision (float, 32-бита)

D – double precision (double, 64-бита)

Скалярные SSE/AVX-инструкции

- **Скалярные SSE-инструкции** (scalar instruction) – в операции участвуют только младшие элементы данных (скаляры) в векторных регистрах/памяти
- ADDSS, SUBSS, MULSS, DIVSS, ADDSD, SUBSD, MULSD, DIVSD, SQRTSS, RSQRTSS, RCPSS, MAXSS, MINSS, ...

Scalar Single-precision (float)				
XMM0	4.0	3.0	2.0	1.0
XMM1	7.0	7.0	7.0	7.0

addss %xmm0, %xmm1

XMM1	4.0	3.0	2.0	8.0
------	-----	-----	-----	-----

- Результат помещается в младшее двойное слово (32-bit) операнда-назначения (xmm1)
- Три старших двойных слова из операнда-источника (xmm0) копируются в операнд-назначение (xmm1)

Scalar Double-precision (double)		
XMM0	8.0	6.0
XMM1	7.0	7.0

addsd %xmm0, %xmm1

XMM1	8.0	13.0
------	-----	------

- Результат помещается в младшие 64 бита операнда-назначения (xmm1)
- Старшие 64 бита из операнда-источника (xmm0) копируются в операнд-назначение (xmm1)

Инструкции над упакованными векторами

- **SSE-инструкция над упакованными векторами** (packed instruction) – в операции участвуют все элементы векторных регистров/памяти
- ADDPS, SUBPS, MULPS, DIVPS, ADDPD, SUBPD, MULPD, DIVPD, SQRTPS, RSQRTPS, RCPPS, MAXPS, MINPS, ...

Packed Single-precision (float)				
XMM0	4.0	3.0	2.0	1.0
XMM1	7.0	7.0	7.0	7.0
addps %xmm0, %xmm1				
XMM1	11.0	10.0	9.0	8.0

Packed Double-precision (double)		
XMM0	8.0	6.0
XMM1	7.0	7.0
addpd %xmm0, %xmm1		
XMM1	15.0	13.0

Инструкции

- **Операции копирования данных (mem-reg/reg-mem/reg-reg)**

- Scalar: MOVSS
- Packed: MOVAPS, MOVUPS, MOVLPS, MOVHPS, MOVLHPS, MOVHLPS

- **Арифметические операции**

- Scalar: ADDSS, SUBSS, MULSS, DIVSS, RCPSS, SQRTSS, MAXSS, MINSS, RSQRTSS
- Packed: ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, MAXPS, MINPS, RSQRTPS

- **Операции сравнения**

- Scalar: CMPSS, COMISS, UCOMISS
- Packed: CMPPS

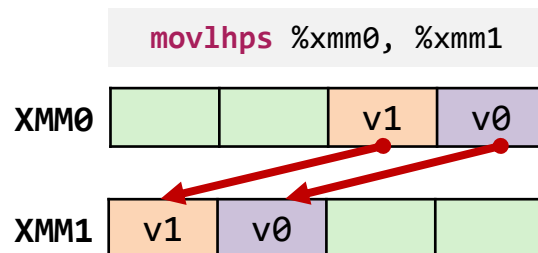
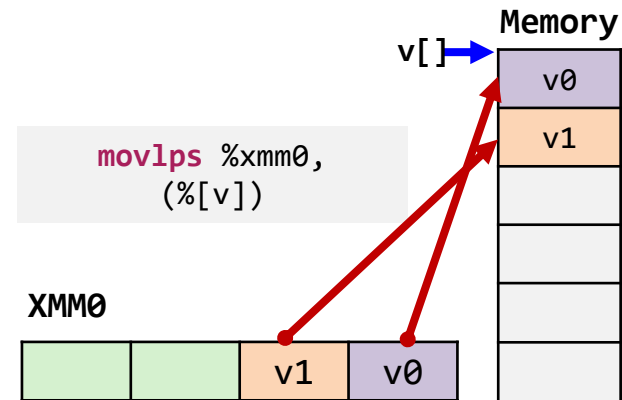
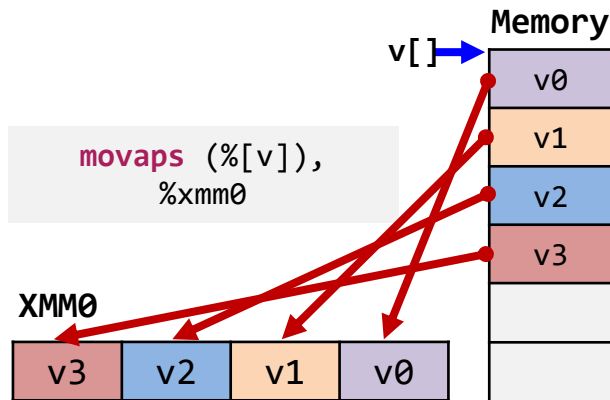
- **Поразрядные логические операции**

- Packed: ANDPS, ORPS, XORPS, ANDNPS

- ...

Arithmetic	Scalar Operator	Packed Operator
$y = y + x$	addss	addps
$y = y - x$	subss	subps
$y = y \times x$	mulss	mulps
$y = y \div x$	divss	divps
$y = \frac{1}{x}$	rcpss	rcpps
$y = \sqrt{x}$	sqrtss	sqrtps
$y = \frac{1}{\sqrt{x}}$	rsqrtss	rsqrtps
$y = \max(y, x)$	maxss	maxps
$y = \min(y, x)$	minss	minps

SSE-инструкции копирования данных



Использование инструкций SSE



Сложение векторов

```
void add(float *a, float *b, float *c)
{
    int i;

    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Вставка на ассемблере

```
void add_sse_asm(float *a, float *b, float *c)
{
    __asm__ __volatile__
    (
        "movaps (%[a]), %%xmm0 \n\t"
        "movaps (%[b]), %%xmm1 \n\t"
        "addps %%xmm1, %%xmm0 \n\t"
        "movaps %%xmm0, %[c] \n\t"
        : [c] "=m" (*c)           /* output */
        : [a] "r" (a), [b] "r" (b) /* input */
        : "%xmm0", "%xmm1"        /* modified regs */
    );
}
```


SSE Intrinsics (builtin functions)

- **Intrinsics** – набор встроенных функций и типов данных, поддерживаемых компилятором, для предоставления высокоуровневого доступа к SSE-инструкциям
- Компилятор самостоятельно распределяет XMM/YMM регистры, принимает решение о способе загрузки данных из памяти (проверяет выравнен адрес или нет) и т.п.
- **Заголовочные файлы:**

```
#include <mmintrin.h>    /* MMX */
#include <xmmmintrin.h>   /* SSE, нужен также mmintrin.h */
#include <emmintrin.h>    /* SSE2, нужен также xmmmintrin.h */
#include <pmmmintrin.h>   /* SSE3, нужен также emmintrin.h */
#include <smmmintrin.h>   /* SSE4.1 */
#include <nmmmintrin.h>   /* SSE4.2 */
#include <immintrin.h>    /* AVX */
```

SSE Intrinsics: типы данных

```
void main()
{
    __m128  f;    /* float[4] */
    __m128d d;    /* double[2] */
    __m128i i;    /* char[16], short int[8], int[4], uint64_t [2] */
}
```

`_mm_<intrinsic_name>_<suffix>`

```
{
    float v[4] = {1.0, 2.0, 3.0, 4.0};
    __m128 t1 = _mm_load_ps(v); // v must be 16-byte aligned

    __m128 t2 = _mm_set_ps(4.0, 3.0, 2.0, 1.0);
}
```

Сложение векторов: SSE intrinsics

```
#include <xmmintrin.h>    /* SSE */

void add(float *a, float *b, float *c)
{
    __m128 t0, t1;

    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

Выравнивание адресов памяти: Microsoft Windows

■ Выравнивание памяти

Хранимые в памяти операнды SSE-инструкций должны быть размещены по адресу выровненному на границу в 16 байт

```
/* Определение статического массива */
__declspec(aligned(16)) float A[N];

/*
 * Динамическое выделение памяти
 * с заданным выравниванием адреса
 */
#include <malloc.h>
void *_aligned_malloc(size_t size, size_t alignment);
void _aligned_free(void *memblock);
```

Выравнивание адресов памяти: GNU/Linux

```
/* Определение статического массива */
float A[N] __attribute__((aligned(16)));


/*
 * Динамическое выделение памяти
 * с заданным выравниванием адреса
 */
#include <malloc.h>
void *_mm_malloc(size_t size, size_t align)
void _mm_free(void *p)

#include <stdlib.h>
int posix_memalign(void **memptr, size_t alignment, size_t size);

/* C11 */
#include <stdlib.h>
void *aligned_alloc(size_t alignment, size_t size);
```

Intel Intrinsics Guide

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

 **Intrinsics Guide**

The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code.

?

<input type="checkbox"/> MMX	<code>__m256d _mm256_broadcast_pd (__m128d const * mem_addr)</code>	<code>vbroadcastf128</code>
<input type="checkbox"/> SSE	<code>__m256 _mm256_broadcast_ps (__m128 const * mem_addr)</code>	<code>vbroadcastf128</code>
<input type="checkbox"/> SSE2	<code>__m256d _mm256_broadcast_sd (double const * mem_addr)</code>	<code>vbroadcastsd</code>
<input type="checkbox"/> SSE3	<code>__m128 _mm_broadcast_ss (float const * mem_addr)</code>	<code>vbroadcastss</code>
<input type="checkbox"/> SSSE3	<code>__m256 _mm256_broadcast_ss (float const * mem_addr)</code>	<code>vbroadcastss</code>
<input type="checkbox"/> SSE4.1		
<input type="checkbox"/> SSE4.2		
<input checked="" type="checkbox"/> AVX		
<input type="checkbox"/> AVX2	<code>__m256i _mm256_lddqu_si256 (__m256i const * mem_addr)</code>	<code>vlddqu</code>
<input type="checkbox"/> FMA	<code>__m256d _mm256_load_pd (double const * mem_addr)</code>	<code>vmovapd</code>
<input type="checkbox"/> AVX-512	<code>__m256 _mm256_load_ps (float const * mem_addr)</code>	<code>vmovaps</code>
<input type="checkbox"/> KNC	<code>__m256i _mm256_load_si256 (__m256i const * mem_addr)</code>	<code>vmovdqa</code>
<input type="checkbox"/> SVM	<code>__m256d _mm256_loadu_pd (double const * mem_addr)</code>	<code>vmovupd</code>
<input type="checkbox"/> Other	<code>__m256 _mm256_loadu_ps (float const * mem_addr)</code>	<code>vmovups</code>
	<code>__m256i _mm256_loadu_si256 (__m256i const * mem_addr)</code>	<code>vmovdqu</code>
	<code>__m256 _mm256_loadu2_m128 (float const* hiaddr, float const* loaddr)</code>	<code>...</code>
	<code>__m256d _mm256_loadu2_m128d (double const* hiaddr, double const* loaddr)</code>	<code>...</code>
	<code>__m256i _mm256_loadu2_m128i (__m128i const* hiaddr, __m128i const* loaddr)</code>	<code>...</code>
	<code>__m128d _mm_maskload_pd (double const * mem_addr, __m128i mask)</code>	<code>vmaskmovpd</code>
	<code>__m256d _mm256_maskload_pd (double const * mem_addr, __m256i mask)</code>	<code>vmaskmovpd</code>
	<code>__m128 _mm_maskload_ps (float const * mem_addr, __m128i mask)</code>	<code>vmaskmovps</code>
	<code>__m256 _mm256_maskload_ps (float const * mem_addr, __m256i mask)</code>	<code>vmaskmovps</code>

Technologies

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☒ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX-512
- ☐ KNC
- ☐ SVM
- ☐ Other

Categories

- ☐ Application-Targeted
- ☐ Arithmetic
- ☐ Bit Manipulation
- ☐ Cast
- ☐ Compare
- ☐ Convert
- ☐ Cryptography
- ☐ Elementary Math

Functions

- ☐ General Support
- ☒ Load
- ☐ Logical
- ☐ Mask

Инициализация векторов

t	4.0	3.0	2.0	1.0	<code>t = _mm_set_ps(4.0, 3.0, 2.0, 1.0);</code>
t	1.0	1.0	1.0	1.0	<code>t = _mm_set1_ps(1.0);</code>
t	0.0	0.0	0.0	1.0	<code>t = _mm_set_ss(1.0);</code>
t	0.0	0.0	0.0	0.0	<code>t = _mm_setzero_ps();</code>

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

Арифметические операции

```
#include <xmmintrin.h>    /* SSE */
```

Intrinsic Name	Operation	Corresponding SSE Instruction
<code>_mm128 _mm_add_ss(__m128 a, __m128 b)</code>	Addition	ADDSS
<code>_mm_add_ps</code>	Addition	ADDPS
<code>_mm_sub_ss</code>	Subtraction	SUBSS
<code>_mm_sub_ps</code>	Subtraction	SUBPS
<code>_mm_mul_ss</code>	Multiplication	MULSS
<code>_mm_mul_ps</code>	Multiplication	MULPS
<code>_mm_div_ss</code>	Division	DIVSS
<code>_mm_div_ps</code>	Division	DIVPS
<code>_mm_sqrt_ss</code>	Squared Root	SQRTSS
<code>_mm_sqrt_ps</code>	Squared Root	SQRTPS
<code>_mm_rcp_ss</code>	Reciprocal	RCPSS
<code>_mm_rcp_ps</code>	Reciprocal	RCPPS
<code>_mm_rsqrt_ss</code>	Reciprocal Squared Root	RSQRTSS
<code>_mm_rsqrt_ps</code>	Reciprocal Squared Root	RSQRTPS
<code>_mm_min_ss</code>	Computes Minimum	MINSS
<code>_mm_min_ps</code>	Computes Minimum	MINPS
<code>_mm_max_ss</code>	Computes Maximum	MAXSS
<code>_mm_max_ps</code>	Computes Maximum	MAXPS

Арифметические операции

```
#include <emmintrin.h>    /* SSE2 */
```

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
__m128d _mm_add_sd(__m128d a, __m128d b)	Addition	ADDSD
_mm_add_pd	Addition	ADDPD
_mm_sub_sd	Subtraction	SUBSD
_mm_sub_pd	Subtraction	SUBPD
_mm_mul_sd	Multiplication	MULSD
_mm_mul_pd	Multiplication	MULPD
_mm_div_sd	Division	DIVSD
_mm_div_pd	Division	DIVPD
_mm_sqrt_sd	Computes Square Root	SQRTSD
_mm_sqrt_pd	Computes Square Root	SQRTPD
_mm_min_sd	Computes Minimum	MINSD
_mm_min_pd	Computes Minimum	MINPD
_mm_max_sd	Computes Maximum	MAXSD
_mm_max_pd	Computes Maximum	MAXPD

Intel® C++ Compiler XE 13.1 User and Reference Guides

SAXPY: scalar version

```
enum { n = 1000000 };

void saxpy(float *x, float *y, float a, int n)
{
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}

double run_scalar()
{
    float *x, *y, a = 2.0;
    x = xmalloc(sizeof(*x) * n);
    y = xmalloc(sizeof(*y) * n);
    for (int i = 0; i < n; i++) {
        x[i] = i * 2 + 1.0;
        y[i] = i;
    }

    double t = wtime();
    saxpy(x, y, a, n);
    t = wtime() - t;

    /* Verification ... */

    printf("Elapsed time (scalar): %.6f sec.\n", t);
    free(x); free(y);
    return t;
}
```

SAXPY
Scalar Alpha X Product Y

$$Y[i] = a * X[i] + Y[i]$$

SAXPY: SSE version

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

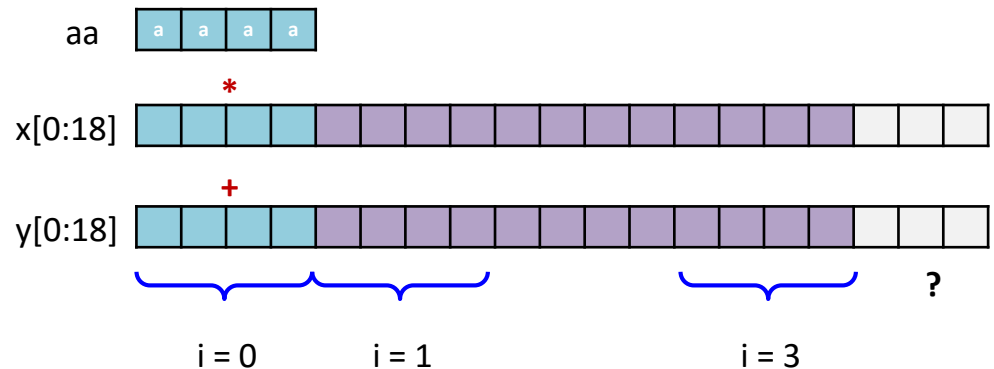
```
void saxpy_sse(float * restrict x, float * restrict y, float a, int n)
```

```
{
    __m128 *xx = (__m128 *)x;
    __m128 *yy = (__m128 *)y;

    int k = n / 4;
    __m128 aa = _mm_set1_ps(a);
    for (int i = 0; i < k; i++) {
        __m128 z = _mm_mul_ps(aa, xx[i]);
        yy[i] = _mm_add_ps(z, yy[i]);
    }
}
```

```
double run_vectorized()
```

```
{
    float *x, *y, a = 2.0;
    x = _mm_malloc(sizeof(*x) * n, 16);
    y = _mm_malloc(sizeof(*y) * n, 16);
    for (int i = 0; i < n; i++) {
        x[i] = i * 2 + 1.0;
        y[i] = i;
    }
    double t = wtime();
    saxpy_sse(x, y, a, n);
    t = wtime() - t;
}
```

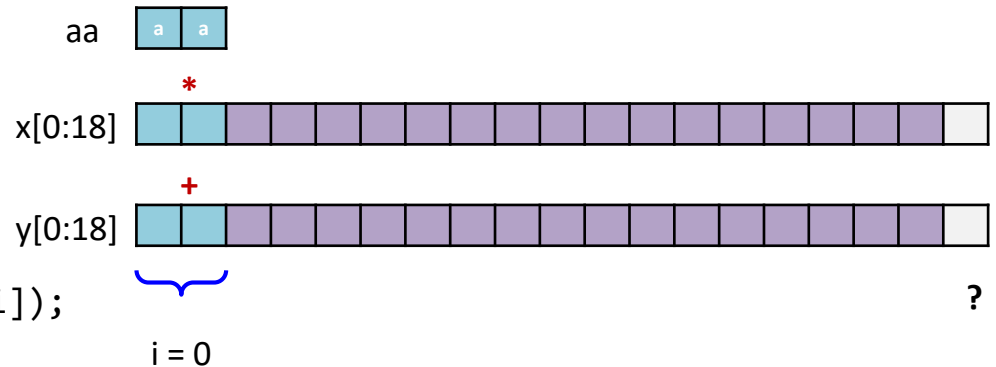


```
# Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
$ ./saxpy
SAXPY (y[i] = a * x[i] + y[i]; n = 1000000)
Elapsed time (scalar): 0.001571 sec.
Elapsed time (vectorized): 0.000835 sec.
Speedup: 1.88
```

DAXPY: SSE version (double precision)

```
void daxpy_sse(double * restrict x, double * restrict y, double a, int n)
{
    __m128d *xx = (__m128d *)x;
    __m128d *yy = (__m128d *)y;

    int k = n / 2;
    __m128d aa = _mm_set1_pd(a);
    for (int i = 0; i < k; i++) {
        __m128d z = _mm_mul_pd(aa, xx[i]);
        yy[i] = _mm_add_pd(z, yy[i]);
    }
}
```



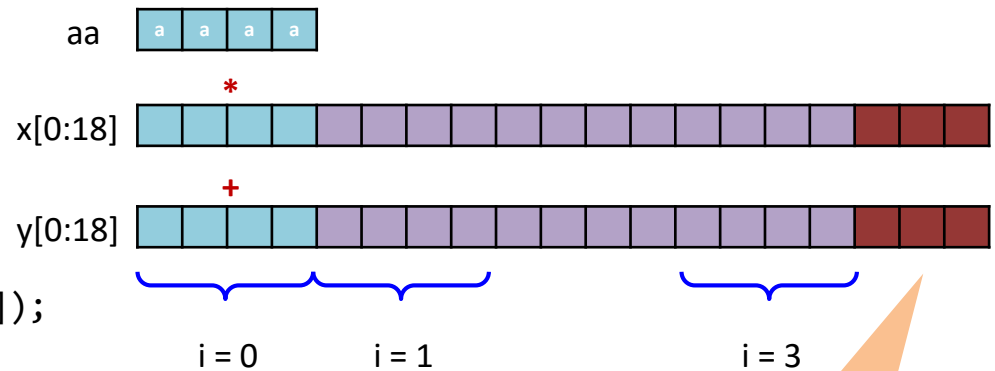
```
# Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
$ ./daxpy
daxpy (y[i] = a * x[i] + y[i]; n = 1000000)
Elapsed time (scalar): 0.002343 sec.
Elapsed time (vectorized): 0.001728 sec.
Speedup: 1.36
```

SAXPY: SSE version + «докрутка» цикла

```
void saxpy_sse(float * restrict x, float * restrict y, float a, int n)
{
    __m128 *xx = (__m128 *)x;
    __m128 *yy = (__m128 *)y;

    int k = n / 4;
    __m128 aa = _mm_set1_ps(a);
    for (int i = 0; i < k; i++) {
        __m128 z = _mm_mul_ps(aa, xx[i]);
        yy[i] = _mm_add_ps(z, yy[i]);
    }

    for (int i = k * 4; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```



SAXPY: AVX version

```
#include <immintrin.h>
```

```
void saxpy_avx(float * restrict x, float * restrict y, float a, int n)
```

```
{
```

```
    __m256 *xx = (__m256 *)x;
```

```
    __m256 *yy = (__m256 *)y;
```

```
    int k = n / 8;
```

```
    __m256 aa = _mm256_set1_ps(a);
```

```
    for (int i = 0; i < k; i++) {
```

```
        __m256 z = _mm256_mul_ps(aa, xx[i]);
```

```
        yy[i] = _mm256_add_ps(z, yy[i]);
```

```
    }
```

```
    for (int i = k * 8; i < n; i++)
```

```
        y[i] = a * x[i] + y[i];
```

```
}
```

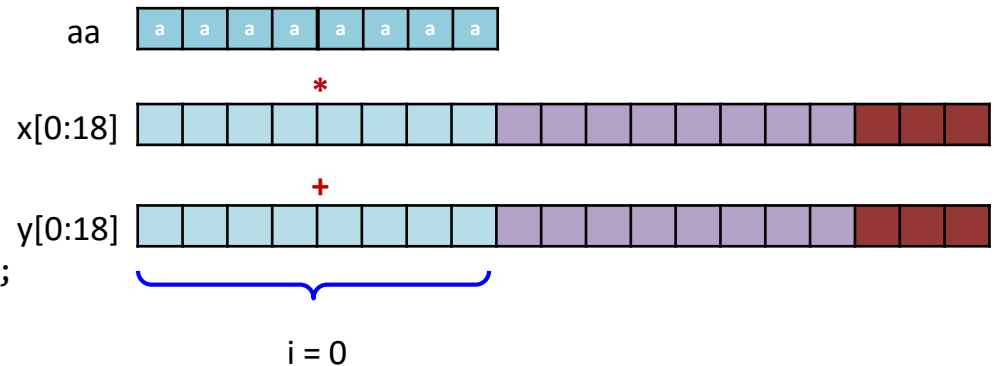
```
double run_vectorized()
```

```
{
```

```
    float *x, *y, a = 2.0;
```

```
    x = _mm_malloc(sizeof(*x) * n, 32);
```

```
    ...
```



Суммирование элементов массива

```
enum {  
    n = 1000014  
};  
  
float sum(float *v, int n)  
{  
    float x = 0.0f;  
    for (int i = 0; i < n; i++)  
        x = x + sqrtf(v[i]);  
    return x;  
}
```

В общем случае неизвестно:

- Выравнен ли адрес *v* на требуемую границу
- Кратна ли длина *n* массива числу элементов *float* в векторном регистре

Цикл разбивается на три (loop splitting):

1. Обработка начальных элементов в скалярном режиме, пока не достигнем элемента с адресом, выравненным на необходимую границу (peeled loop)
2. Векторизованный цикл
3. Обработка «хвоста» (reminder loop)

Суммирование элементов массива

```
#include <immintrin.h>
```

```
#define SIMD_WIDTH_BYTES 32
```

```
#define SIMD_WIDTH_F32 8
```

```
float sum_avx_peeled(float * restrict v, int n)
{
    float x = 0.0f;
    int peeled_tripcount = 0;
    int misalign_bytes = (uintptr_t)v & (SIMD_WIDTH_BYTES - 1);
    if (misalign_bytes > 0) {
        // 1. Peeled loop for proper alignment
        peeled_tripcount = (SIMD_WIDTH_BYTES - misalign_bytes) /
                           sizeof(float);
        for (int i = 0; i < peeled_tripcount; i++) {
            x = x + sqrtf(v[i]);
        }
    }

    // 2. Vectorized loop
    __m256 xx = _mm256_setzero_ps();
    int main_tripcount = n - ((n - peeled_tripcount) & (SIMD_WIDTH_F32 - 1));
    for (int i = peeled_tripcount; i < main_tripcount; i += SIMD_WIDTH_F32) {
        xx = _mm256_add_ps(xx, _mm256_sqrt_ps(_mm256_load_ps(&v[i])));
    }
}
```


Суммирование элементов массива

```
// Horizontal summation xx[0..15]
```

```
xx = _mm256_hadd_ps(xx, xx);
```

```
xx = _mm256_hadd_ps(xx, xx);
```

```
// Permute high and low 128 bits of xx
```

```
xx = _mm256_add_ps(xx, _mm256_permute2f128_ps(xx, xx, 1));
```

```
float tmp;
```

```
_mm_store_ss(&tmp, _mm256_castps256_ps128(xx));
```

```
x = x + tmp;
```

```
// 3. Reminder loop
```

```
for (int i = main_tripcount; i < n; i++) {
```

```
    x = x + sqrtf(v[i]);
```

```
}
```

```
return x;
```

```
}
```

Particles

```
enum { n = 1000003 };
```

```
void init_particles(float *x, float *y, float *z, int n)
{
    for (int i = 0; i < n; i++) {
        x[i] = cos(i + 0.1);
        y[i] = cos(i + 0.2);
        z[i] = cos(i + 0.3);
    }
}
```

```
void distance(float *x, float *y, float *z,
             float *d, int n)
{
    for (int i = 0; i < n; i++) {
        d[i] = sqrtf(x[i] * x[i] + y[i] * y[i] +
                    z[i] * z[i]);
    }
}
```

```
double run_scalar()
```

```
{
    float *d, *x, *y, *z;
    x = xmalloc(sizeof(*x) * n);
    y = xmalloc(sizeof(*y) * n);
    z = xmalloc(sizeof(*z) * n);
    d = xmalloc(sizeof(*d) * n);

    init_particles(x, y, z, n);

    double t = wtime();
    for (int iter = 0; iter < 100; iter++) {
        distance(x, y, z, d, n);
    }
    t = wtime() - t;
    return t;
}
```

Particles: SSE

```
void distance_vec(float *x, float *y, float *z, float *d, int n)
{
    __m128 *xx = (__m128 *)x;
    __m128 *yy = (__m128 *)y;
    __m128 *zz = (__m128 *)z;
    __m128 *dd = (__m128 *)d;

    int k = n / 4;
    for (int i = 0; i < k; i++) {
        __m128 t1 = _mm_mul_ps(xx[i], xx[i]);
        __m128 t2 = _mm_mul_ps(yy[i], yy[i]);
        __m128 t3 = _mm_mul_ps(zz[i], zz[i]);
        t1 = _mm_add_ps(t1, t2);
        t1 = _mm_add_ps(t1, t3);
        dd[i] = _mm_sqrt_ps(t1);
    }

    for (int i = k * 4; i < n; i++) {
        d[i] = sqrtf(x[i] * x[i] + y[i] * y[i] + z[i] * z[i]);
    }
}
```

Particles: AVX

```
void distance_vec(float *x, float *y, float *z, float *d, int n)
{
    __m256 *xx = (__m256 *)x;
    __m256 *yy = (__m256 *)y;
    __m256 *zz = (__m256 *)z;
    __m256 *dd = (__m256 *)d;

    int k = n / 8;
    for (int i = 0; i < k; i++) {
        __m256 t1 = _mm256_mul_ps(xx[i], xx[i]);
        __m256 t2 = _mm256_mul_ps(yy[i], yy[i]);
        __m256 t3 = _mm256_mul_ps(zz[i], zz[i]);
        t1 = _mm256_add_ps(t1, t2);
        t1 = _mm256_add_ps(t1, t3);
        dd[i] = _mm256_sqrt_ps(t1);
    }

    for (int i = k * 8; i < n; i++) {
        d[i] = sqrtf(x[i] * x[i] + y[i] * y[i] + z[i] * z[i]);
    }
}
```

Редукция (reduction, reduce)

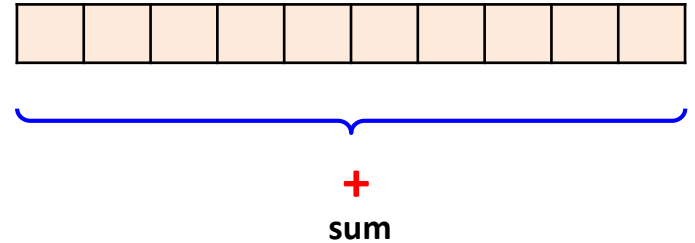
```
enum { n = 1000003 };
```

```
float sum(float *v, int n)
{
    float s = 0;
    for (int i = 0; i < n; i++)
        s += v[i];
    return s;
}
```

```
double run_scalar()
{
    float *v = xmalloc(sizeof(*v) * n);
    for (int i = 0; i < n; i++)
        v[i] = i + 1.0;

    double t = wtime();
    float res = sum(v, n);
    t = wtime() - t;

    float valid_result = (1.0 + (float)n) * 0.5 * n;
    printf("Result (scalar): %.6f err = %f\n", res, fabsf(valid_result - res));
    printf("Elapsed time (scalar): %.6f sec.\n", t);
    free(v);
    return t;
}
```



Редукция (reduction, reduce)

```
enum { n = 1000003 };
```

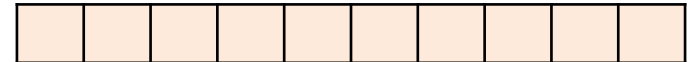
```
float sum(float *v, int n)
{
    float s = 0;
    for (int i = 0; i < n; i++)
        s += v[i];
    return s;
}
```

```
double run_scalar()
```

```
{
    float *v = xmalloc(sizeof(*v) * n);
    for (int i = 0; i < n; i++)
        v[i] = i + 1.0;

    double t = wtime();
    float res = sum(v, n);
    t = wtime() - t;

    float valid_result = (1.0 + (float)n) * 0.5 * n;
    printf("Result (scalar): %.6f err = %f\n", res, fabsf(valid_result - res));
    printf("Elapsed time (scalar): %.6f sec.\n", t);
    free(v);
    return t;
}
```



+

sum

```
$ ./reduction
```

```
Reduction: n = 1000003
```

```
Result (scalar): 499944423424.000000 err =  
59080704.0
```

```
Elapsed time (scalar): 0.001011 sec.
```

?

Редукция (reduction, reduce)

```
enum { n = 1000003 };
```

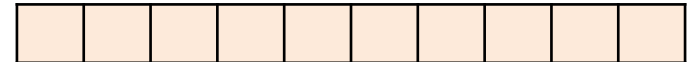
```
float sum(float *v, int n)
{
    float s = 0;
    for (int i = 0; i < n; i++)
        s += v[i];
    return s;
}
```

```
double run_scalar()
```

```
{
    float *v = xmalloc(sizeof(*v) * n);
    for (int i = 0; i < n; i++)
        v[i] = i + 1.0;

    double t = wtime();
    float res = sum(v, n);
    t = wtime() - t;
```

```
float valid_result = (1.0 + (float)n) * 0.5 * n;
printf("Result (scalar): %.6f err = %f\n", res, fabsf(valid_result - res));
printf("Elapsed time (scalar): %.6f sec.\n", t);
free(v);
return t;
}
```



+

sum

- **float** (IEEE 754, single-precision) имеет ограниченную точность
- погрешность результата суммирования n чисел в худшем случае растет пропорционально n

Вариант 1: переход от float к double

```
double sum(double *v, int n)
{
    double s = 0;
    for (int i = 0; i < n; i++)
        s += v[i];
    return s;
}
```

```
double run_scalar()
{
    double *v = xmalloc(sizeof(*v) * n);
    for (int i = 0; i < n; i++)
        v[i] = i + 1.0;

    double t = wtime();
    double res = sum(v, n);
    t = wtime() - t;
```

```
double valid_result = (1.0 + (double)n) * 0.5 * n;
printf("Result (scalar): %.6f err = %f\n", res, fabsf(valid_result - res));
printf("Elapsed time (scalar): %.6f sec.\n", t);
free(v);
return t;
}
```

```
$ ./reduction
```

```
Reduction: n = 1000003
```

```
Result (scalar): 500003500006.000000 err = 0.000000
```

```
Elapsed time (scalar): 0.001031 sec.
```


Вариант 2: компенсационное суммирование Кэхэна

```
float sum(float *v, int n)
{
    float s = 0;
    for (int i = 0; i < n; i++)
        s += v[i];
    return s;
}
```



```
/*
 * Алгоритм Кэхэна (Kahan's summation) -- компенсационное
 * суммирование чисел с плавающей запятой в формате IEEE 754 [*]
 *
 * [*] Kahan W. Further remarks on reducing truncation errors //
 * Communications of the ACM - 1964 - Vol. 8(1). - P. 40.
 */
float sum_kahan(float *v, int n)
{
    float s = v[0];
    float c = (float)0.0;

    for (int i = 1; i < n; i++) {
        float y = v[i] - c;
        float t = s + y;
        c = (t - s) - y;
        s = t;
    }
    return s;
}
```

погрешность
не зависит от n
(только от точности float)

- **W. M. Kahan** – один из основных разработчиков IEEE 754
(Turing Award-1989, ACM Fellow) <http://www.cs.berkeley.edu/~wkahan>

Вариант 2: компенсационное суммирование Кэхэна

```
float sum(float *v, int n)
{
    float s = 0;
    for (int i = 0; i < n; i++)
        s += v[i];
    return s;
}
```



```
/*
 * Алгоритм Кэхэна (Kahan's summation) -- компенсационное
 * суммирование чисел с плавающей запятой в формате IEEE 754 [*]
 *
 * [*] Kahan W. Further remarks on reducing truncation errors //
 * Communications of the ACM - 1964 - Vol. 8(1). - P. 40.
 */
float sum_kahan(float *v, int n)
{
    float s = v[0];
    float c = (float)0.0;

    for (int i = 1; i < n; i++) {
        float y = v[i] - c;
        float t = s + y;
        c = (t - s) - y;
        s = t;
    }
    return s;
}
```

погрешность
не зависит от n
(только от точности float)

```
$ ./reduction
Reduction: n = 1000003
Result (scalar): 500003504128.000000 err = 0.000000
Elapsed time (scalar): 0.004312 sec.
```

Векторная версия редукции: SSE, float

```
double run_vectorized()
{
    float *v = _mm_malloc(sizeof(*v) * n, 16);
    for (int i = 0; i < n; i++)
        v[i] = 2.0;

    double t = wtime();
    float res = sum_sse(v, n);
    t = wtime() - t;

    float valid_result = 2.0 * (float)n;
    printf("Result (vectorized): %.6f err = %f\n", res, fabsf(valid_result - res));
    printf("Elapsed time (vectorized): %.6f sec.\n", t);
    free(v);
    return t;
}
```

Векторная версия редукции: SSE, float

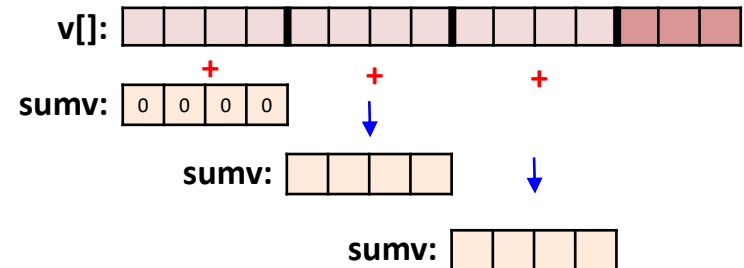
```
float sum_sse(float * restrict v, int n)
{
    __m128 *vv = (__m128 *)v;
    int k = n / 4;
    __m128 sumv = _mm_setzero_ps();
    for (int i = 0; i < k; i++) {
        sumv = _mm_add_ps(sumv, vv[i]);
    }
```

```
// s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
float t[4] __attribute__((aligned(16)));
_mm_store_ps(t, sumv);
float s = t[0] + t[1] + t[2] + t[3];
```

```
for (int i = k * 4; i < n; i++)
    s += v[i];
return s;
```

```
}
```

1) Векторное суммирование (вертикальное)



2) Горизонтальное суммирование

`s = sumv[0] + sumv[1] + sumv[2] + sumv[3]`

3) Скалярное суммирование элементов в «хвосте» массива

Векторная версия редукции: SSE, float

```
float sum_sse(float * restrict v, int n)
{
    __m128 *vv = (__m128 *)v;
    int k = n / 4;
    __m128 sumv = _mm_setzero_ps();
    for (int i = 0; i < k; i++) {
        sumv = _mm_add_ps(sumv, vv[i]);
    }

    // s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
    float t[4] __attribute__((aligned(16)));
    _mm_store_ps(t, sumv);
    float s = t[0] + t[1] + t[2] + t[3];
}
```

```
# cngpu1: Intel Core i5 4690 - Haswell
Reduction: n = 1000003
Result (scalar): 2000006.000000 err = 0.000000
Elapsed time (scalar): 0.003862 sec.
Result (vectorized): 2000006.000000 err = 0.000000
Elapsed time (vectorized): 0.002523 sec.
Speedup: 1.53
```

```
# Intel Core i5-3320M - Ivy Bridge (Sandy Bridge shrink)
Reduction: n = 1000003
Result (scalar): 2000006.000000 err = 0.000000
Elapsed time (scalar): 0.001074 sec.
Result (vectorized): 2000006.000000 err = 0.000000
Elapsed time (vectorized): 0.000760 sec.
Speedup: 1.41
```

```
# Oak: Intel Xeon E5620 - Westmere (Nehalem shrink)
Reduction: n = 1000003
Result (scalar): 2000006.000000 err = 0.000000
Elapsed time (scalar): 0.001259 sec.
Result (vectorized): 2000006.000000 err = 0.000000
Elapsed time (vectorized): 0.000315 sec.
Speedup: 3.99
```

```
# cnmic: Intel Xeon E5-2620 v3 - Haswell
Reduction: n = 1000003
Result (scalar): 2000006.000000 err = 0.000000
Elapsed time (scalar): 0.001256 sec.
Result (vectorized): 2000006.000000 err = 0.000000
Elapsed time (vectorized): 0.000319 sec.
Speedup: 3.94
```

Векторная версия редукции: погрешность вычислений

```
double run_vectorized()
{
    float *v = _mm_malloc(sizeof(*v) * n, 16);
    for (int i = 0; i < n; i++)
        v[i] = i + 1.0;    // Изменили инициализацию с "2.0" на "i + 1.0"

    double t = wtime();
    float res = sum_sse(v, n);
    t = wtime() - t;

    float valid_result = (1.0 + (float)n) * 0.5 * n;
    printf("Result (vectorized): %.6f err = %f\n", res, fabsf(valid_result - res));
    printf("Elapsed time (vectorized): %.6f sec.\n", t);
    free(v);
    return t;
}
```

Результаты скалярной
и векторной версий
не совпадают!

```
$ ./reduction
Reduction: n = 1000003
Result (scalar): 499944423424.000000 err = 59080704.000000
Elapsed time (scalar): 0.001007 sec.
Result (vectorized): 500010975232.000000 err = 7471104.000000
Elapsed time (vectorized): 0.000770 sec.
Speedup: 1.31
```

Векторная версия редукции: погрешность вычислений

- **Скалярная версия:**

$s = v[0] + v[1] + v[2] + \dots + v[n - 1]$

- **Векторная SSE-версия (float):**

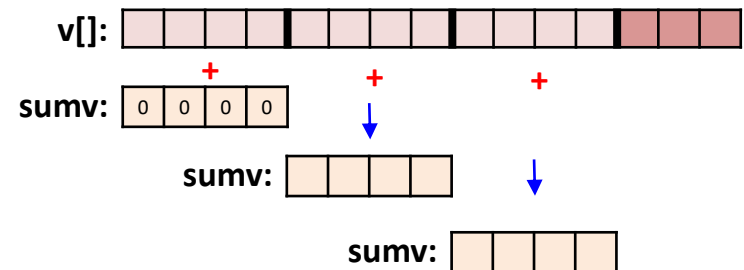
```
s = (v[0] + v[4] + v[8]) + // sumv[0]
    (v[1] + v[5] + v[9]) + // sumv[1]
    (v[2] + v[6] + v[10]) + // sumv[2]
    (v[3] + v[7] + v[11]) + // sumv[3]
    v[12] + v[13] + v[14] // «хвост»
```

- В SSE-версии порядок выполнения операций отличается от скалярной версии
- Операция сложения чисел с плавающей запятой в формате IEEE 754 не ассоциативна и не коммутативна

$$a + b \neq b + a \quad a + (b + c) \neq (a + b) + c$$

- David Goldberg. *What Every Computer Scientist Should Know About Floating-Point Arithmetic* // <http://www.validlab.com/goldberg/paper.pdf>

1) Векторное суммирование (вертикальное)



2) Горизонтальное суммирование

$s = \text{sumv}[0] + \text{sumv}[1] + \text{sumv}[2] + \text{sumv}[3]$

3) Скалярное суммирование элементов в «хвосте» массива

Векторная версия редукции: SSE, double

```
double sum_sse(double * restrict v, int n)
{
    __m128d *vv = (__m128d *)v;
    int k = n / 2;
    __m128d sumv = _mm_setzero_pd();
    for (int i = 0; i < k; i++) {
        sumv = _mm_add_pd(sumv, vv[i]);
    }
    // Compute s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
    double t[2] __attribute__((aligned(16)));
    _mm_store_pd(t, sumv);
    double s = t[0] + t[1];

    for (int i = k * 2; i < n; i++)
        s += v[i];
    return s;
}
```

```
$ ./reduction
Reduction: n = 1000003
Result (scalar): 500003500006.000000 err = 0.000000
Elapsed time (scalar): 0.001134 sec.
Result (vectorized): 500003500006.000000 err = 0.000000
Elapsed time (vectorized): 0.001525 sec.
Speedup: 0.74
```


Векторная версия: горизонтальное суммирование SSE3

```
#include <pmmintrin.h>
```

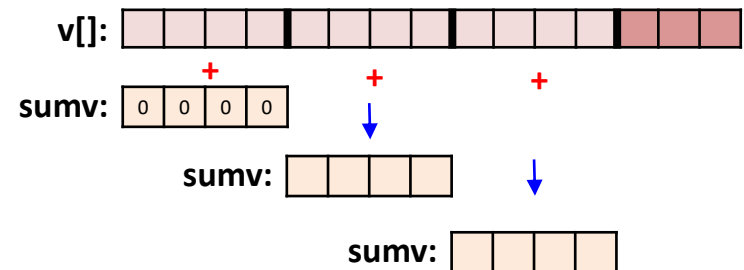
```
float sum_sse(float * restrict v, int n)
{
    __m128 *vv = (__m128 *)v;
    int k = n / 4;
    __m128 sumv = _mm_setzero_ps();
    for (int i = 0; i < k; i++) {
        sumv = _mm_add_ps(sumv, vv[i]);
    }
}
```

```
// s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
sumv = _mm_hadd_ps(sumv, sumv);
sumv = _mm_hadd_ps(sumv, sumv);
float s __attribute__((aligned (16))) = 0;
_mm_store_ss(&s, sumv);
```

```
for (int i = k * 4; i < n; i++)
    s += v[i];
return s;
```

```
}
```

1) Векторное суммирование (вертикальное)



2) Горизонтальное суммирование SSE3

$a = \text{hadd}(a, a) \Rightarrow a = [a3 + a2 \mid a1 + a0 \mid a3 + a2 \mid a1 + a0]$
 $a = \text{hadd}(a, a) \Rightarrow a = [a3 + a2 + a1 + a0 \mid \text{---} \mid \text{---} \mid \text{---}]$

3) Скалярное суммирование элементов в «хвосте» массива

Векторная версия: горизонтальное суммирование SSE3

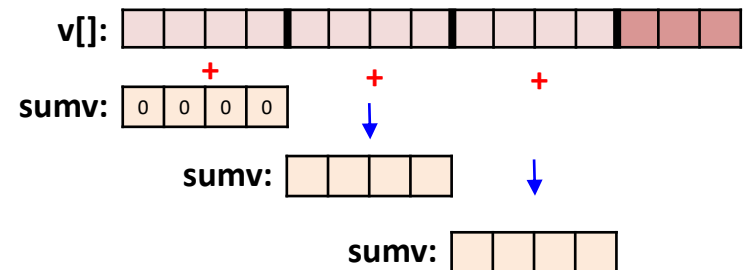
```
#include <pmmmintrin.h>
```

```
float sum_sse(float * restrict v, int n)
{
    __m128 *vv = (__m128 *)v;
    int k = n / 4;
    __m128 sumv = _mm_setzero_ps();
    for (int i = 0; i < k; i++) {
        sumv = _mm_add_ps(sumv, vv[i]);
    }
}
```

```
// s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
sumv = _mm_hadd_ps(sumv, sumv);
sumv = _mm_hadd_ps(sumv, sumv);
float s __attribute__((aligned (16))) = 0;
_mm_store_ss(&s, sumv);
```

```
for (int i = k * 4; i < n; i++)
    s += v[i];
return s;
}
```

1) Векторное суммирование (вертикальное)



2) Горизонтальное суммирование SSE3

`a = hadd(a, a) => a = [a3 + a2 | a1 + a0 | a3 + a2 | a1 + a0]`
`a = hadd(a, a) => a = [a3 + a2 + a1 + a0 | --/-- | --/-- | --/--]`

```
# Intel Core i5-3320M - Ivy Bridge (Sandy Bridge shrink)
Reduction: n = 1000003
Result (scalar): 499944423424.000000 err = 59080704.000000
Elapsed time (scalar): 0.001071 sec.
Result (vectorized): 500010975232.000000 err = 7471104.000000
Elapsed time (vectorized): 0.000342 sec.
Speedup: 3.13
```

Векторная версия: горизонтальное суммирование (double)

```
double sum_sse(double * restrict v, int n)
{
    __m128d *vv = (__m128d *)v;
    int k = n / 2;
    __m128d sumv = _mm_setzero_pd();
    for (int i = 0; i < k; i++) {
        sumv = _mm_add_pd(sumv, vv[i]);
    }

    // Compute s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
    // SSE3 horizontal operation:
    //   hadd(a, a) => a = [a1 + a0 | a1 + a0]
    sumv = _mm_hadd_pd(sumv, sumv);
    double s __attribute__((aligned(16))) = 0;
    _mm_store_sd(&s, sumv);

    for (int i = k * 2; i < n; i++)
        s += v[i];
    return s;
}
```

```
# Intel Core i5-3320M - Ivy Bridge (Sandy Bridge shrink)
Reduction: n = 1000003
Result (scalar): 500003500006.000000 err = 0.000000
Elapsed time (scalar): 0.001047 sec.
Result (vectorized): 500003500006.000000 err = 0.000000
Elapsed time (vectorized): 0.000636 sec.
Speedup: 1.65
```

Векторная версия: AVX (double)

```
double run_vectorized()
{
    double *v = _mm_malloc(sizeof(*v) * n, 32);
    for (int i = 0; i < n; i++)
        v[i] = i + 1.0;

    double t = wtime();
    double res = sum_avx(v, n);
    t = wtime() - t;

    double valid_result = (1.0 + (double)n) * 0.5 * n;
    printf("Result (vectorized): %.6f err = %f\n", res, fabsf(valid_result - res));
    printf("Elapsed time (vectorized): %.6f sec.\n", t);
    free(v);
    return t;
}
```

Векторная версия: AVX (double)

```
#include <immintrin.h>
double sum_avx(double * restrict v, int n)
{
    __m256d *vv = (__m256d *)v;
    int k = n / 4;
    __m256d sumv = _mm256_setzero_pd();
    for (int i = 0; i < k; i++) {
        sumv = _mm256_add_pd(sumv, vv[i]);
    }

    // Compute s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
    // AVX _mm256_hadd_pd:
    //   _mm256_hadd_pd(a, a) => a = [a3 + a2 | a3 + a2 | a1 + a0 | a1 + a0]
    sumv = _mm256_hadd_pd(sumv, sumv);
    // Permute high and low 128 bits of sumv: [a1 + a0 | a1 + a0 | a3 + a2 | a3 + a2]
    __m256d sumv_permuted = _mm256_permute2f128_pd(sumv, sumv, 1);
    // sumv = [a1 + a0 + a3 + a2 | --/-- | ...]
    sumv = _mm256_add_pd(sumv_permuted, sumv);

    double t[4] __attribute__((aligned (16)));
    _mm256_store_pd(t, sumv);
    double s = t[0]; //double s = t[0] + t[1] + t[2] + t[3];
    for (int i = k * 4; i < n; i++)
        s += v[i];
    return s;
}
```

Векторная версия: AVX (double)

```
#include <immintrin.h>
double sum_avx(double * restrict v, int n)
{
    __m256d *vv = (__m256d *)v;
    int k = n / 4;
    __m256d sumv = _mm256_setzero_pd();
    for (int i = 0; i < k; i++) {
        sumv = _mm256_add_pd(sumv, vv[i]);
    }

    // Compute s = sumv[0] + sumv[1] + sumv[2] + sumv[3]
    // AVX _mm256_hadd_pd:
    //   _mm256_hadd_pd(a, a) => a = [a3 + a2 | a3 + a2 | a1 + a0 | a1 + a0]
    sumv = _mm256_hadd_pd(sumv, sumv);
    // Permute high and low 128 bits of sumv: [a1 + a0 | a1 + a0 | a3 + a2 | a3 + a2]
    __m256d sumv_permuted = _mm256_permute2f128_pd(sumv, sumv, 1);
    // sumv = [a1 + a0 + a3 + a2 | --/-- | ...]
    sumv = _mm256_add_pd(sumv_permuted, sumv);

    double t[4] __attribute__((aligned(16)));
    _mm256_store_pd(t, sumv);
    double s = t[0]; //double s = t[0] + t[1] + t[2]
    for (int i = k * 4; i < n; i++)
        s += v[i];
    return s;
}
```

```
# Intel Core i5-3320M - Ivy Bridge (Sandy Bridge shrink)
Reduction: n = 1000003
Result (scalar): 500003500006.000000 err = 0.000000
Elapsed time (scalar): 0.001061 sec.
Result (vectorized): 500003500006.000000 err = 0.000000
Elapsed time (vectorized): 0.000519 sec.
Speedup: 2.04
```

Поиск максимума в массиве: скалярная версия

```
float find_max(float *v, int n)
{
    float max = -FLT_MAX;
    for (int i = 0; i < n; i++)
        if (v[i] > max)
            max = v[i];
    return max;
}

double run_scalar()
{
    float *v = xmalloc(sizeof(*v) * n);
    for (int i = 0; i < n; i++)
        v[i] = i + 1.0;

    double t = wtime();
    float res = find_max(v, n);
    t = wtime() - t;

    float valid_result = (float)n;
    printf("Result (scalar): %.6f err = %f\n", res, fabsf(valid_result - res));
    printf("Elapsed time (scalar): %.6f sec.\n", t);
    free(v);
    return t;
}
```

Поиск максимума в массиве: SSE, float

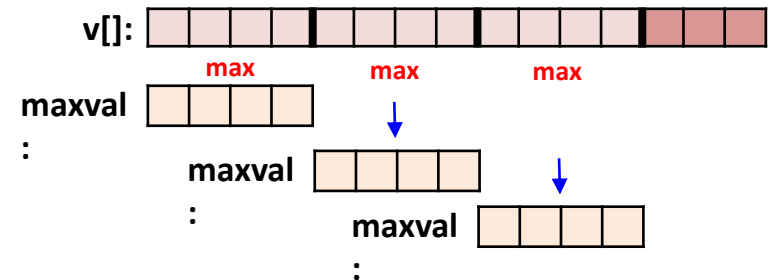
```
float find_max_sse(float * restrict v, int n)
{
    __m128 *vv = (__m128 *)v;
    int k = n / 4;

    __m128 maxval = _mm_set1_ps(-FLT_MAX);
    for (int i = 0; i < k; i++)
        maxval = _mm_max_ps(maxval, vv[i]);

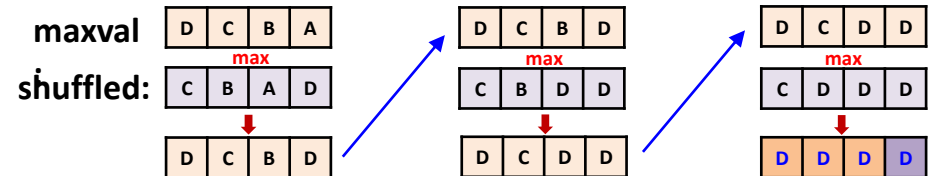
    // Horizontal max
    // a = [a3, a2, a1, a0]
    // shuffle(a, a, _MM_SHUFFLE(2, 1, 0, 3)) ==> [a2, a1, a0, a3]
    maxval = _mm_max_ps(maxval, _mm_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm_max_ps(maxval, _mm_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm_max_ps(maxval, _mm_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    float max;
    _mm_store_ss(&max, maxval);

    for (int i = k * 4; i < n; i++)
        if (v[i] > max)
            max = v[i];
    return max;
}
```

1) Векторный поиск максимума (вертикальная операция)



2) Горизонтальный поиск максимума в векторе



3) Скалярный поиск максимума в «хвосте» массива

Поиск максимума в массиве: SSE, float

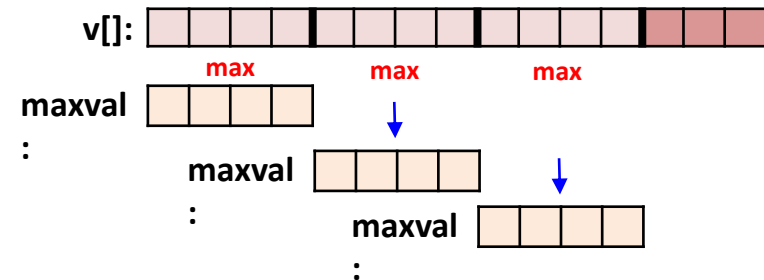
```
float find_max_sse(float * restrict v, int n)
{
    __m128 *vv = (__m128 *)v;
    int k = n / 4;

    __m128 maxval = _mm_set1_ps(-FLT_MAX);
    for (int i = 0; i < k; i++)
        maxval = _mm_max_ps(maxval, vv[i]);

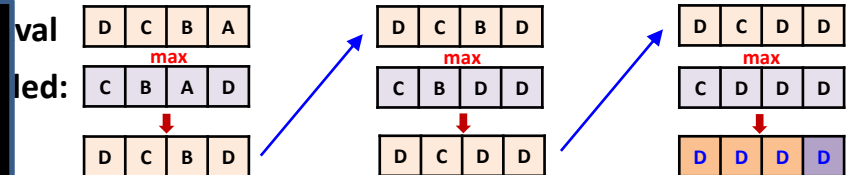
    // Horizontal max
    // a = [a3, a2, a1, a0]
    // shuffle(a, a, _MM_SHUFFLE(2, 1, 0, 3)) ==> [a2, a1, a0, a3]
    maxval = _mm_max_ps(maxval, _mm_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm_max_ps(maxval, _mm_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm_max_ps(maxval, _mm_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    float max;
    _mm_store_ss(&max, maxval);
}
```

```
# Intel Core i5-3320M - Ivy Bridge (Sandy Bridge shrink)
Reduction: n = 1000003
Result (scalar): 1000003.000000 err = 0.000000
Elapsed time (scalar): 0.002047 sec.
Result (vectorized): 1000003.000000 err = 0.000000
Elapsed time (vectorized): 0.000529 sec.
Speedup: 3.87
```

1) Векторный поиск максимума (вертикальная операция)



2) Горизонтальный поиск максимума в векторе



3) Скалярный поиск максимума в «хвосте» массива

Поиск максимума в массиве: AVX, float

```
float find_max_avx(float * restrict v, int n)
{
    __m256 *vv = (__m256 *)v;
    int k = n / 8;

    __m256 maxval = _mm256_set1_ps(-FLT_MAX);
    for (int i = 0; i < k; i++)
        maxval = _mm256_max_ps(maxval, vv[i]);

    maxval = _mm256_max_ps(maxval, _mm256_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm256_max_ps(maxval, _mm256_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm256_max_ps(maxval, _mm256_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    float t[8];
    _mm256_store_ps(t, maxval);
    float max = t[0] > t[5] ? t[0] : t[5];

    for (int i = k * 8; i < n; i++)
        if (v[i] > max)
            max = v[i];
    return max;
}
```

1) Горизонтальная операция над двумя частями по 128 бит
[a7, a6, a5, a4 | a3, a2, a1, a0] ==> [a6, a5, a4, a7 | a2, a1, a0, a3]
[a6, a5, a4, a7 | a2, a1, a0, a3] ==> [a5, a4, a7, a6 | a1, a0, a3, a2]
[a5, a4, a7, a6 | a1, a0, a3, a2] ==> [a4, a7, a6, a5 | a0, a3, a2, a1]
[a4, a7, a6, a5 | a0, a3, a2, a1]

2) Выбор максимального из t[5] и t[0]

3) Обработка «хвоста»

Поиск максимума в массиве: AVX, float

```
float find_max_avx(float * restrict v, int n)
{
    __m256 *vv = (__m256 *)v;
    int k = n / 8;

    __m256 maxval = _mm256_set1_ps(-FLT_MAX);
    for (int i = 0; i < k; i++)
        maxval = _mm256_max_ps(maxval, vv[i]);

    maxval = _mm256_max_ps(maxval, _mm256_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm256_max_ps(maxval, _mm256_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm256_max_ps(maxval, _mm256_shuffle_ps(maxval, maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    float t[8];
    _mm256_store_ps(t, maxval);
    float max = t[0] > t[5] ? t[0] : t[5];

    for (int i = k * 8; i < n; i++)
        if (v[i] > max)
            max = v[i];
}
```

1) Горизонтальная операция над двумя частями по 128 бит

[a7, a6, a5, a4 | a3, a2, a1, a0] ==> [a6, a5, a4, a7 | a2, a1, a0, a3]

[a6, a5, a4, a7 | a2, a1, a0, a3] ==> [a5, a4, a7, a6 | a1, a0, a3, a2]

[a5, a4, a7, a6 | a1, a0, a3, a2] ==> [a4, a7, a6, a5 | a0, a3, a2, a1]

Intel Core i5-3320M - Ivy Bridge (Sandy Bridge shrink)

Reduction: n = 1000003

Result (scalar): 1000003.000000 err = 0.000000

Elapsed time (scalar): 0.002053 sec.

Result (vectorized): 1000003.000000 err = 0.000000

Elapsed time (vectorized): 0.000428 sec.

Speedup: 4.80

[a6, a5 | a0, a3, a2, a1]

о максимального из t[5] и t[0]

ботка «хвоста»

Поиск максимума в массиве: AVX, float, **permute**

```
float find_max_avx(float * restrict v, int n)
{
    __m256 *vv = (__m256 *)v;
    int k = n / 8;

    __m256 maxval = _mm256_set1_ps(-FLT_MAX);
    for (int i = 0; i < k; i++)
        maxval = _mm256_max_ps(maxval, vv[i]);

    // Horizontal max
    maxval = _mm256_max_ps(maxval, _mm256_permute_ps(maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm256_max_ps(maxval, _mm256_permute_ps(maxval, _MM_SHUFFLE(2, 1, 0, 3)));
    maxval = _mm256_max_ps(maxval, _mm256_permute_ps(maxval, _MM_SHUFFLE(2, 1, 0, 3)));

    float t[8];
    _mm256_store_ps(t, maxval);
    float max = t[0] > t[5] ? t[0] : t[5];

    for (int i = k * 8; i < n; i++)
        if (v[i] > max)
            max = v[i];
    return max;
}
```

```
# Intel Core i5-3320M - Ivy Bridge (Sandy Bridge shrink)
Reduction: n = 1000003
Result (scalar): 1000003.000000 err = 0.000000
Elapsed time (scalar): 0.002201 sec.
Result (vectorized): 1000003.000000 err = 0.000000
Elapsed time (vectorized): 0.000433 sec.
Speedup: 5.08
```

Вычисление квадратного корня: скалярная версия

```
void compute_sqrt(float *in, float *out, int n)
{
    for (int i = 0; i < n; i++) {
        if (in[i] > 0)
            out[i] = sqrtf(in[i]);
        else
            out[i] = 0.0;
    }
}
```

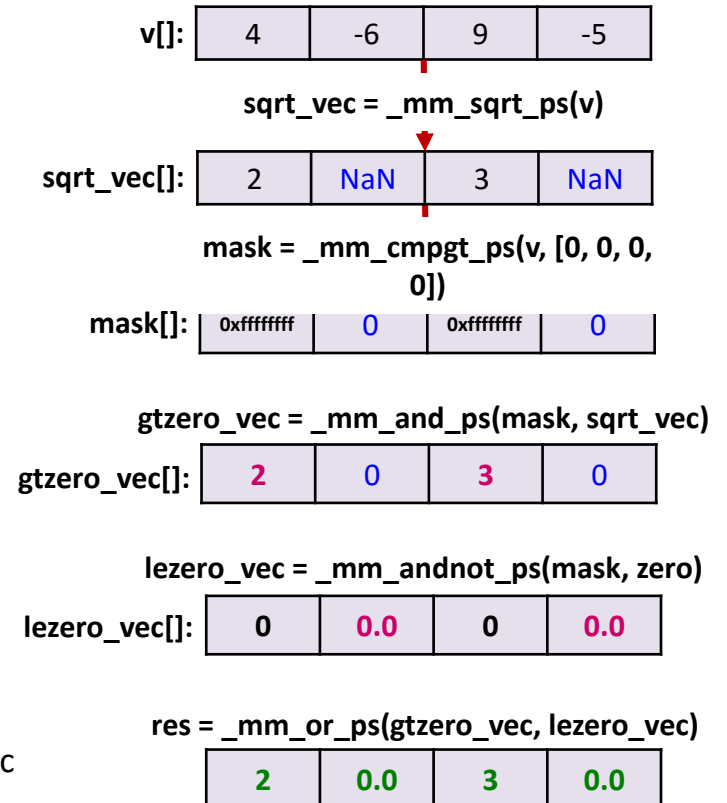
Направление ветвления в цикле
зависит от входных данных

```
double run_scalar()
{
    float *in = xmalloc(sizeof(*in) * n);
    float *out = xmalloc(sizeof(*out) * n);
    srand(0);
    for (int i = 0; i < n; i++) {
        in[i] = rand() > RAND_MAX / 2 ? 0 : rand() / (float)RAND_MAX * 1000.0;
    }
    double t = wtime();
    compute_sqrt(in, out, n);
    t = wtime() - t;

    printf("Elapsed time (scalar): %.6f sec.\n", t);
    free(in);
    free(out);
    return t;
}
```

Вычисление квадратного корня: SSE, float

1. **Вычисляем корень квадратный для вектора $v[0:3]$** –
если значение $v[i]$ меньше или равно нулю, результат NaN
`sqrt_vec = _mm_sqrt_ps(v)`
2. **Выполняем векторное сравнение:** $v[0:3] > [0, 0, 0, 0]$
`mask = _mm_cmpgt_ps(v, zero)`
результат сравнения – вектор `mask[0:3]`,
в котором `mask[i] = v[i] > 0 ? 0xffffffff : 0`
3. **Извлекаем из `sqrt_vec[0:3]` элементы**, для которых выполнено
условие $v[i] > 0$
`gtzero_vec = _mm_and_ps(mask, sqrt_vec)`
В `gtzero_vec` элементы NaN заменены на 0
4. **Извлекаем из `zero[0:3]` элементы**, для которых условие
не выполнено: $v[i] \leq 0$
`lezero_vec = _mm_andnot_ps(mask, zero)`
В `lezero_vec` элементы 0 заменены на 0.0 (значение в ветви else).
5. **Объединяем результаты ветвей** – векторы `gtzero_vec` и `lezero_vec`
`res = _mm_or_ps(gtzero_vec, lezero_vec)`

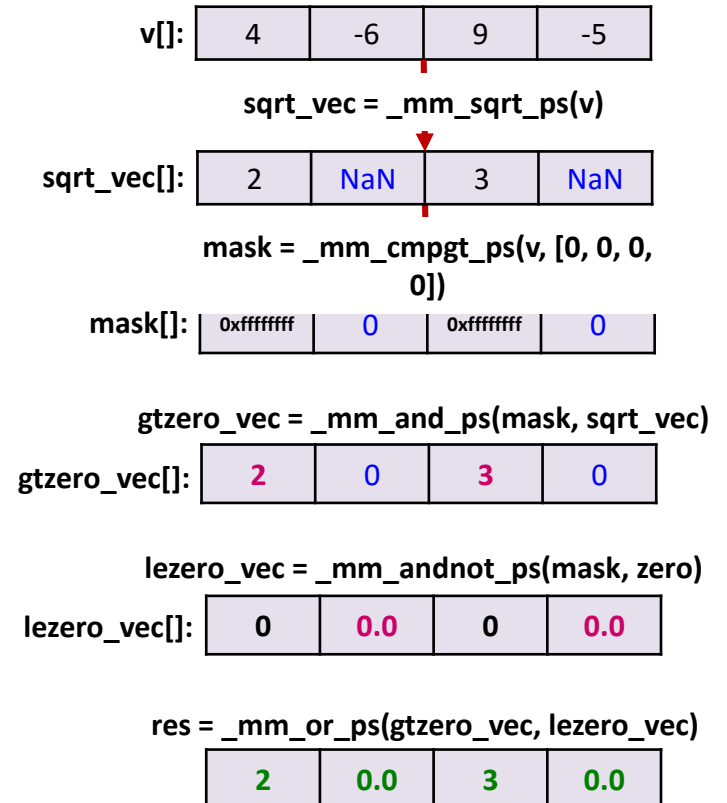


Вычисление квадратного корня: SSE, float

```
void compute_sqrt_sse(float *in, float *out, int n)
{
    __m128 *in_vec = (__m128 *)in;
    __m128 *out_vec = (__m128 *)out;
    int k = n / 4;

    __m128 zero = _mm_setzero_ps();
    for (int i = 0; i < k; i++) {
        __m128 v = _mm_load_ps((float *)&in_vec[i]);
        __m128 sqrt_vec = _mm_sqrt_ps(v);
        __m128 mask = _mm_cmpgt_ps(v, zero);
        __m128 gtzero_vec = _mm_and_ps(mask, sqrt_vec);
        __m128 lezero_vec = _mm_andnot_ps(mask, zero);
        out_vec[i] = _mm_or_ps(gtzero_vec, lezero_vec);
    }

    for (int i = k * 4; i < n; i++)
        out[i] = in[i] > 0 ? sqrtf(in[i]) : 0.0;
}
```



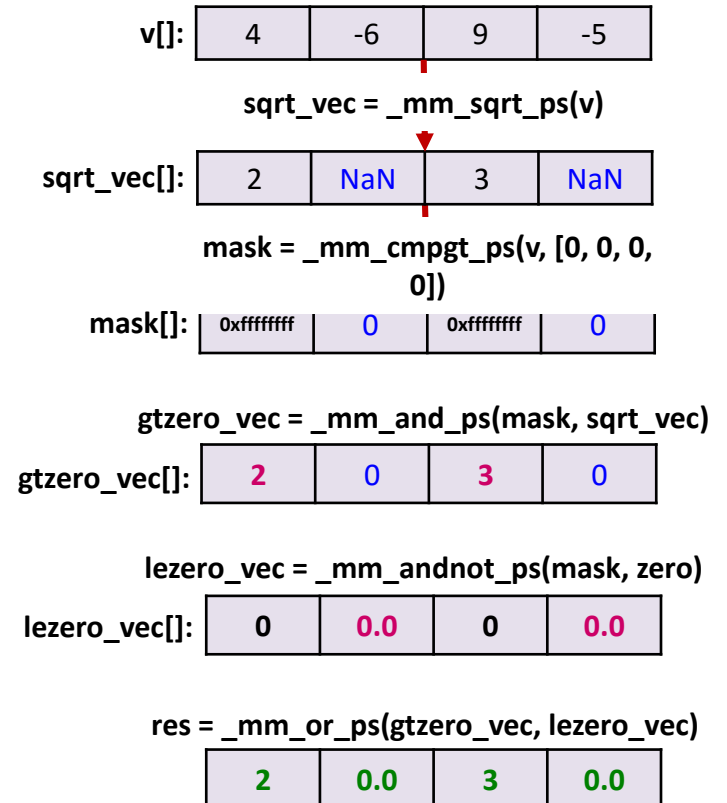
Вычисление квадратного корня: SSE, float

```
void compute_sqrt_sse(float *in, float *out, int n)
{
    __m128 *in_vec = (__m128 *)in;
    __m128 *out_vec = (__m128 *)out;
    int k = n / 4;

    __m128 zero = _mm_setzero_ps();
    for (int i = 0; i < k; i++) {
        __m128 v = _mm_load_ps((float *)&in_vec[i]);
        __m128 sqrt_vec = _mm_sqrt_ps(v);
        __m128 mask = _mm_cmpgt_ps(v, zero);
        __m128 gtzero_vec = _mm_and_ps(mask, sqrt_vec);
        __m128 lezero_vec = _mm_andnot_ps(mask, zero);
        out_vec[i] = _mm_or_ps(gtzero_vec, lezero_vec);
    }

    for (int i = k * 4; i < n; i++)
```

```
# Intel Core i5-3320M - Ivy Bridge (Sandy Bridge shrink)
Tabulate sqrt: n = 1000003
Elapsed time (scalar): 0.009815 sec.
Elapsed time (vectorized): 0.002176 sec.
Speedup: 4.51
```



Вычисление квадратного корня: **AVX**, float

```
void compute_sqrt_avx(float *in, float *out, int n)
{
    __m256 *in_vec = (__m256 *)in;
    __m256 *out_vec = (__m256 *)out;
    int k = n / 8;

    __m256 zero = _mm256_setzero_ps();
    for (int i = 0; i < k; i++) {
        __m256 v = _mm256_load_ps((float *)&in_vec[i]);
        __m256 sqrt_vec = _mm256_sqrt_ps(v);
        __m256 mask = _mm256_cmp_ps(v, zero, _CMP_GT_OQ);
        __m256 gtzero_vec = _mm256_and_ps(mask, sqrt_vec);
        __m256 lezero_vec = _mm256_andnot_ps(mask, zero);
        out_vec[i] = _mm256_or_ps(gtzero_vec, lezero_vec);
    }

    for (int i = k * 8; i < n; i++)
        out[i] = in[i] > 0 ? sqrtf(in[i]) : 0.0;
}
```

Вычисление квадратного корня: AVX, float, blend

```
void compute_sqrt_avx(float *in, float *out, int n)
{
    __m256 *in_vec = (__m256 *)in;
    __m256 *out_vec = (__m256 *)out;
    int k = n / 8;

    __m256 zero = _mm256_setzero_ps();
    for (int i = 0; i < k; i++) {
        // 1. Compute sqrt: all elements <= 0 will be filled with NaNs
        // 2. Vector compare (greater-than): in[i] > 0
        //    mask = cmpgt([7, 1, 0, 2, 0, 4, 4, 9], zero) ==>
        //    mask = [0xffffffff, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff, ..., 0xffffffff]
        //
        // 3. Blend (merge results from two vectors by mask)
        //    blend(zero, [7, 1, 0, 2, 0, 4, 4, 9], mask) = [7, 1, 0f, 2, 0f, 4, 4, 9]
        //
        __m256 v = _mm256_load_ps((float *)&in_vec[i]);
        __m256 sqrt_vec = _mm256_sqrt_ps(v);
        __m256 mask = _mm256_cmp_ps(v, zero, _CMP_GT_OQ);
        out_vec[i] = _mm256_blendv_ps(zero, sqrt_vec, mask);
    }
    for (int i = k * 8; i < n; i++)
        out[i] = in[i] > 0 ? sqrtf(in[i]) : 0.0;
}
```

Автоматическая векторизация кода компилятором

Автоматическая векторизация: GCC 5.3.1

```
enum { n = 1000003 };

int main(int argc, char **argv)
{
    float *a = malloc(sizeof(*a) * n);
    float *b = malloc(sizeof(*b) * n);
    float *c = malloc(sizeof(*c) * n);

    for (int i = 0; i < n; i++) {
        a[i] = 1.0;
        b[i] = 2.0;
    }

    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }

    free(c); free(b); free(a);
    return 0;
}
```

```
$ gcc -Wall -std=c99 -O2 -march=native -ftree-vectorize -fopt-info-vec -c vec.c -o vec.o
vec.c:19:5: note: loop vectorized
vec.c:14:5: note: loop vectorized
gcc -o vec vec.o -lm
```

Автоматическая векторизация: clang 3.7

```
enum { n = 1000003 };

int main(int argc, char **argv)
{
    float *a = malloc(sizeof(*a) * n); float *b = malloc(sizeof(*b) * n); float *c = malloc(sizeof(*c) * n);

    for (int i = 0; i < n; i++) {
        a[i] = 1.0; b[i] = 2.0;
    }

    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }

    free(c); free(b); free(a);
    return 0;
}
```

```
$ clang -Wall -O2 -Rpass=loop-vectorize -Rpass-missed=loop-vectorize \
    -Rpass-analysis=loop-vectorize -c vec.c -o vec.o
vec.c:14:5: remark: vectorized loop (vectorization width: 4, interleaved count: 2) [-Rpass=loop-vectorize]
    for (int i = 0; i < n; i++) {
    ^
vec.c:19:5: remark: vectorized loop (vectorization width: 4, interleaved count: 2) [-Rpass=loop-vectorize]
    for (int i = 0; i < n; i++) {
    ^
$ clang -o vec vec.o
```

Требования к циклам

- **Требования к циклам**

- ☐ Отсутствие зависимости по данным между итерациями цикла
- ☐ Отсутствие вызовов функций в цикле
- ☐ Число итераций цикла должно быть вычислимым
- ☐ Обращение к последовательным смежным элементам массива
- ☐ Отсутствие сложных ветвлений

- **Проблемные ситуации**

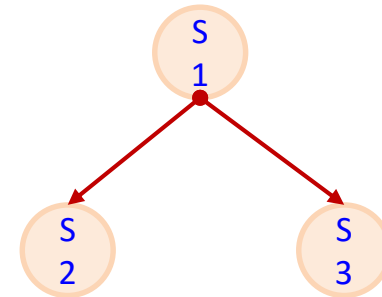
- ☐ Сложный цикл – может не хватить векторных регистров
- ☐ Смешанные типы данных (int, float, char)

Зависимости по данным между инструкциями

S1: $A = B + C$

S2: $D = A + 2$

S3: $E = A + 3$



Граф зависимостей по данным
(data-dependence graph)

- **S2 зависит от S1** – S1 и S2 нельзя выполнять параллельно
- **S3 зависит от S1** – S1 и S3 нельзя выполнять параллельно
- S2 и S3 можно выполнять параллельно



**Векторизация программ.
Теория, методы, реализация**
(сборник статей). – М.: Мир, 1991. – 275 с.

Виды зависимости по данным между инструкциями

1. Потокковая зависимость, истинная зависимость

(Read After Write – RAW, true dependence, flow/data dependency): $S1 \delta S2$

S1: $a = \dots$

S2: $b = a$

2. Антязависимость (Write After Read – WAR, anti-dependence): $S1 \bar{\delta} S2$

S1: $b = a$

S2: $a = \dots$

3. Выходная зависимость (Write After Write – WAW, output dependence): $S1 \delta^o S2$

S1: $a = \dots$

S2: $a = \dots$

Виды зависимости по данным между итерациями циклов

- Имеется две строки программы S1 и S2
- Обозначим:
 - Write(S) – множество ячеек памяти, в которые S осуществляет запись
 - Read(S) – множество ячеек памяти, которые S читает
- **Условия Бернштейна.** Строка S2 зависит от строки S1 тогда и только тогда, когда
$$(\text{Read}(S1) \cap \text{Write}(S2)) \cup (\text{Write}(S1) \cap \text{Read}(S2)) \cup (\text{Write}(S1) \cap \text{Write}(S2)) \neq \emptyset$$

[*] A. J. Bernstein. *Program Analysis for Parallel Processing* // IEEE Trans. on Electronic Computers, 1966.

- **Развертка цикла по итерациям:**
 - S1: $a[1] = a[0] + b[0]$
 - S2: $a[2] = a[1] + b[1]$
- $\text{Read}(S1) = \{a[0], b[0]\}, \text{Write}(S1) = \{a[1]\}$
- $\text{Read}(S2) = \{a[1], b[1]\}, \text{Write}(S2) = \{a[2]\}$
$$(\text{Read}(S1) \cap \text{Write}(S2)) \cup (\text{Write}(S1) \cap \text{Read}(S2)) \cup (\text{Write}(S1) \cap \text{Write}(S2)) = \emptyset \cup \{a[1]\} \cup \emptyset = \{a[1]\}$$

Зависимости по данным между итерациями циклов

```
for (int i = 0; i < n; i++) {  
    a[i] = 1.0;  
    b[i] = 2.0;  
}  
  
S1: for (int i = 0; i < n - 1; i++) {  
    a[i + 1] = a[i] + b[i];  
}
```

- Развертка цикла по итерациям (строка S1):

S1: a[1] = a[0] + b[0]

S1: a[2] = a[1] + b[1] // Read After Write dep.

S1: a[3] = a[2] + b[2] // Read After Write dep.

S1: a[4] = a[3] + b[3] // Read After Write dep.

```
$ gcc -ftree-vectorize -fopt-info-vec -fopt-info-vec-missed ./vec.c  
vec.c:18:5: note: not vectorized, possible dependence between data-refs vec.c:18:5: note: bad data dependence.  
vec.c:18:5: note: not vectorized, possible dependence between data-refs vec.c:18:5: note: bad data dependence.  
...  
vec.c:13:5: note: loop vectorized  
...
```

```
$ clang -Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize ./vec.c -ovec  
./vec.c:13:5: remark: vectorized loop (vectorization width: 4, interleaved count: 2) [-Rpass=loop-vectorize]  
./vec.c:19:20: remark: loop not vectorized: value that could not be identified as reduction is used outside the loop  
[-Rpass-analysis=loop-vectorize]  
./vec.c:18:5: remark: loop not vectorized: use -Rpass-analysis=loop-vectorize for more info  
[-Rpass-missed=loop-vectorize]
```

Виды зависимости по данным между итерациями циклов

1. Read After Write (RAW, true dependence, flow/data dependency)

S1: **a** = ...

S2: b = **a**

2. Write After Read (WAR, anti-dependence)

S1: b = **a**

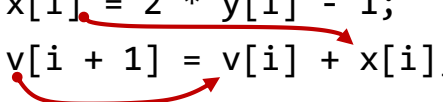
S2: **a** = ...

3. Write After Write (WAW, output dependence)

S1: **a** = ...

S2: **a** = ...

```
S1:   for (int i = 0; i < n - 1; i++) {  
      x[i] = 2 * y[i] - 1;  
S2:   v[i + 1] = v[i] + x[i];  
      }
```



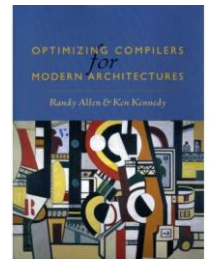
S1 --> S2 (RAW- flow dependence) –
цикло-независимая зависимость

S2 --> S2 (RAW) – *циклическая зависимость*
(требуется выполнить не менее одной итерации
для ее возникновения)

Виды зависимости по данным между итерациями циклов

Утверждение. Цикл может быть векторизован тогда и только тогда, когда в нем отсутствуют циклические зависимости между операциями [*].

[*] Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*.
- Morgan Kaufmann Publishers, 2001.



- Предполагается наличие векторных регистров бесконечной длины (VL)
- Циклические зависимости, для возникновения которых требуется не менее $VL + 1$ итераций, могут быть проигнорированы (например, 5 итераций для SSE-инструкций типа float)

S2:

```
for (int i = 0; i < n - 1; i++) {  
    v[i] = v[i + 3] + 4;  
}
```

Iter 1: $v[0] = v[3] + 4$
Iter 2: $v[1] = v[4] + 4$
Iter 3: $v[2] = v[5] + 4$
Iter 4: $v[3] = v[6] + 4$
Iter 5: $v[4] = v[7] + 4$
Iter 6: $v[5] = v[8] + 4$
...

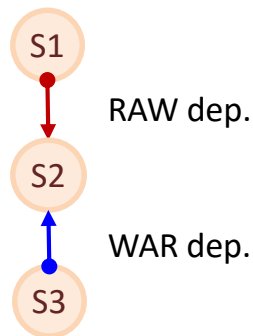
Можно
векторизовать для
(SSE, double),
но не для (SSE, float)!

Зависимость через 3
итерации

Автоматическая векторизация циклов

- Компилятор анализирует граф зависимостей по данным для самых внутренних циклов
- Если в графе зависимостей по данным отсутствуют контуры (замкнутые пути), то его можно векторизовать

```
for (int i = 0; i < n; i++) {  
S1:  a[i] = b[i];  
S2:  c[i] = a[i] + b[i];  
S3:  e[i] = c[i + 1];  
}
```



```
a[0] = b[0]  
c[0] = a[0] + b[0]  
e[0] = c[1]  
  
a[1] = b[1]  
c[1] = a[1] + b[1]  
e[1] = c[2]  
  
a[2] = b[2]  
c[2] = a[2] + b[2]  
e[2] = c[3]  
  
a[3] = b[3]  
c[3] = a[3] + b[3]  
e[3] = c[4]
```

Контуры
отсутствуют



```
// S3 должна предшествовать S2  
a[0:n-1] = b[0:n-1]  
e[0:n-1] = c[1:n]  
c[0:n-1] = a[0:n-1] + b[0:n-1]
```

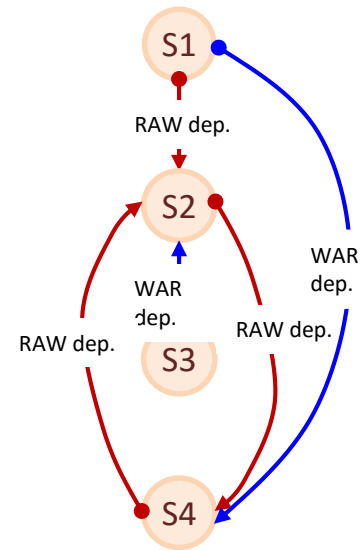
Автоматическая векторизация циклов

```
for (int i = 2; i <= n; i++) {  
S1:    a[i] = b[i];  
S2:    c[i] = a[i] + b[i - 1];  
S3:    e[i] = c[i + 1];  
S4:    b[i] = c[i] + 2;  
}
```

- **Инструкции S2 и S4 образуют контур**
- Все операторы контура исполняются последовательно (...)
- Остальные инструкции векторизуемы



```
a[2:n] = b[2:n]  
e[2:n] = c[3:n + 1]  
for (int i = 2; i <= n; i++) {  
S2:    c[i] = a[i] + b[i - 1];  
S4:    b[i] = c[i] + 2;  
}
```



Инструкции S2 и S4 образуют контур

Автоматическая векторизация циклов

```
void mul_alpha(int *x, int *y, int a, int n)
{
    for (int i = 0; i < n; i++)
        y[i] = a * x[i];
}
```

- Что известно об указателях x и y ?
- Возможно указывают на один массив (пересекаются)
- Компилятору необходимо проводить межпроцедурный анализ или использовать «подсказки»

```
void mul_alpha(int * restrict x, int * restrict y, int a, int n)
{
    for (int i = 0; i < n; i++)
        y[i] = a * x[i];
}
```

- **restrict** – для доступа к объекту используется данный указатель p или значение, основанное на указателе p (например, $p + 1$)

Литература

- Randy Allen, Ken Kennedy. ***Optimizing Compilers for Modern Architectures: A Dependence-Based Approach***. - Morgan Kaufmann Publishers, 2001.
- Steven Muchnick. ***Advanced Compiler Design and Implementation***, 1997
- Aart J.C. Bik. ***Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance***, 2004.
- Keith Cooper, Linda Torczon. ***Engineering a Compiler***, 2011
- ***Векторизация программ. Теория, методы, реализация*** (сборник статей). – М.: Мир, 1991. – 275 с.
- Auto-vectorization in GCC // <https://gcc.gnu.org/projects/tree-ssa/vectorization.html>
- Auto-Vectorization in LLVM // <http://llvm.org/docs/Vectorizers.html>