

# Лекция

# Векторизация кода: SSE/AVX

**Курносов Михаил Георгиевич**

E-mail: [mkurnosov@gmail.com](mailto:mkurnosov@gmail.com)

WWW: [www.mkurnosov.net](http://www.mkurnosov.net)

Курс «Распределенная обработка информации»

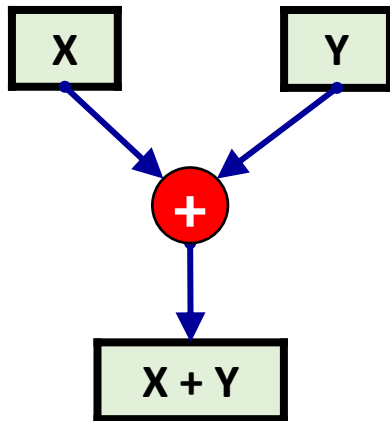
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Осенний семестр, 2020

# Векторные процессоры

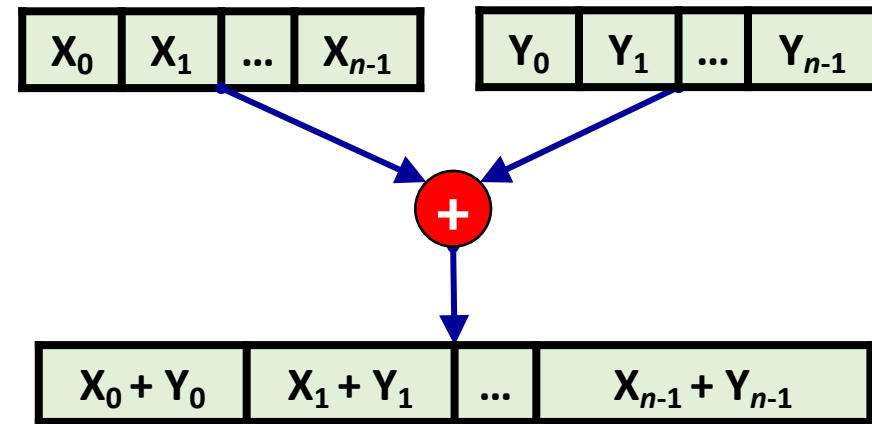
- **Векторный процессор (vector processor)** – процессор, поддерживающий на уровне системы команд операции для работы с одномерными массивами (векторами)

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

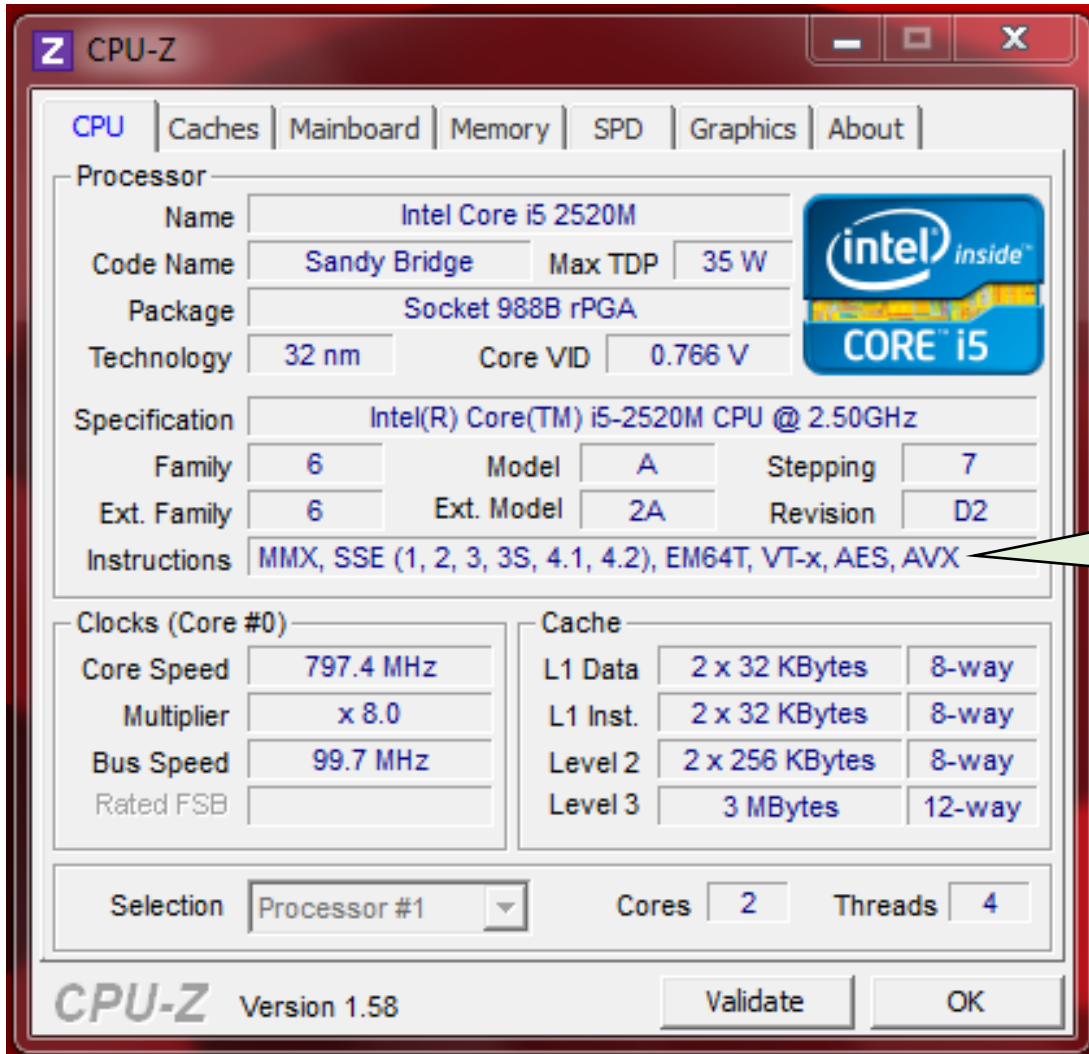
- Cray 1 (1976) 80 MHz, 8 regs, 64 elems
- Cray XMP (1983) 120 MHz 8 regs, 64 elems
- Cray YMP (1988) 166 MHz 8 regs, 64 elems
- Cray C-90 (1991) 240 MHz 8 regs, 128 elems
- Cray T-90 (1996) 455 MHz 8 regs, 128 elems
- Conv. C-1 (1984) 10 MHz 8 regs, 128 elems
- Conv. C-4 (1994) 133 MHz 16 regs, 128 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



The screenshot shows the CPU-Z application window with the 'CPU' tab selected. The processor is identified as an Intel Core i5 2520M, Sandy Bridge architecture, 32 nm technology, with a core voltage of 0.766 V. The specification is Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz. The instructions supported are MMX, SSE (1, 2, 3, 3S, 4.1, 4.2), EM64T, VT-x, AES, and AVX. The clock speed is 797.4 MHz, and the bus speed is 99.7 MHz. The cache configuration includes 2 x 32 KBytes L1 Data and L1 Inst., 2 x 256 KBytes Level 2, and 3 MBytes Level 3. The selection is set to Processor #1, showing 2 cores and 4 threads.

Processor			
Name	Intel Core i5 2520M		
Code Name	Sandy Bridge	Max TDP	35 W
Package	Socket 988B rPGA		
Technology	32 nm	Core VID	0.766 V
Specification	Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz		
Family	6	Model	A
Ext. Family	6	Ext. Model	2A
Stepping	7	Revision	D2
Instructions	MMX, SSE (1, 2, 3, 3S, 4.1, 4.2), EM64T, VT-x, AES, AVX		

Clocks (Core #0)		
Core Speed	797.4 MHz	
Multiplier	x 8.0	
Bus Speed	99.7 MHz	
Rated FSB		

Cache		
L1 Data	2 x 32 KBytes	8-way
L1 Inst.	2 x 32 KBytes	8-way
Level 2	2 x 256 KBytes	8-way
Level 3	3 MBytes	12-way

Selection: Processor #1 | Cores: 2 | Threads: 4

CPU-Z Version 1.58 | Validate | OK

- 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;

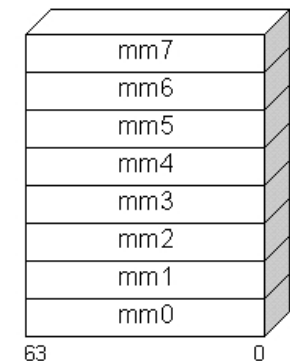
    cpuid(1, &eax, &ebx, &ecx, &edx);
    return (ecx & (1 << 28)) ? 1 : 0;
}

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

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

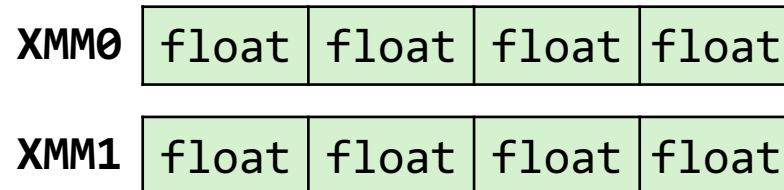
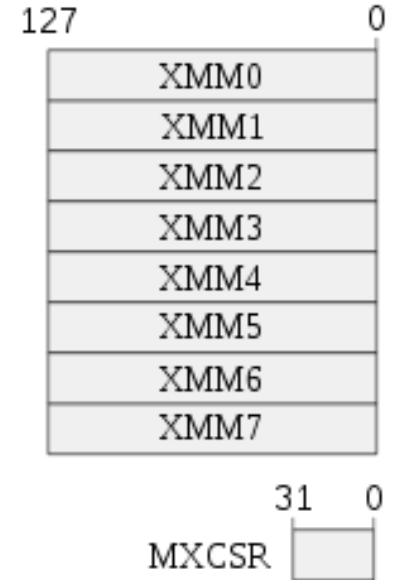
# Intel MMX

- 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 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)

- **Тип элементов вектора/скаляра**

**S** – single precision (float, 32-бита)

**D** – double precision (double, 64-бита)

- **ADDPS** – add 4 packed single-precision values (float)
- **ADDSD** – add 1 scalar double-precision value (double)

# 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

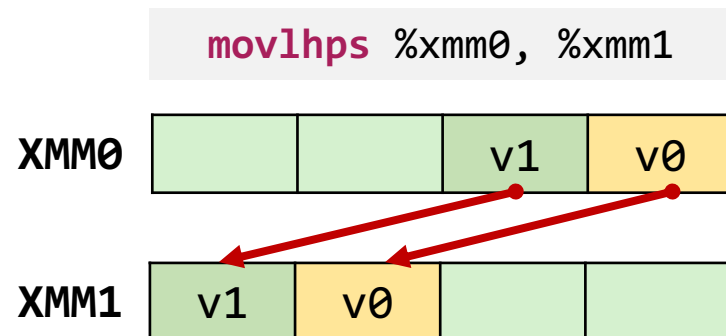
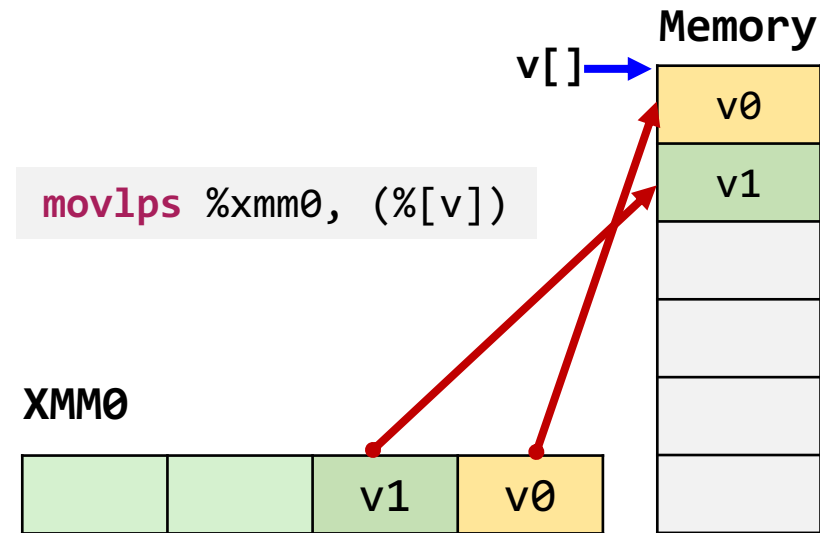
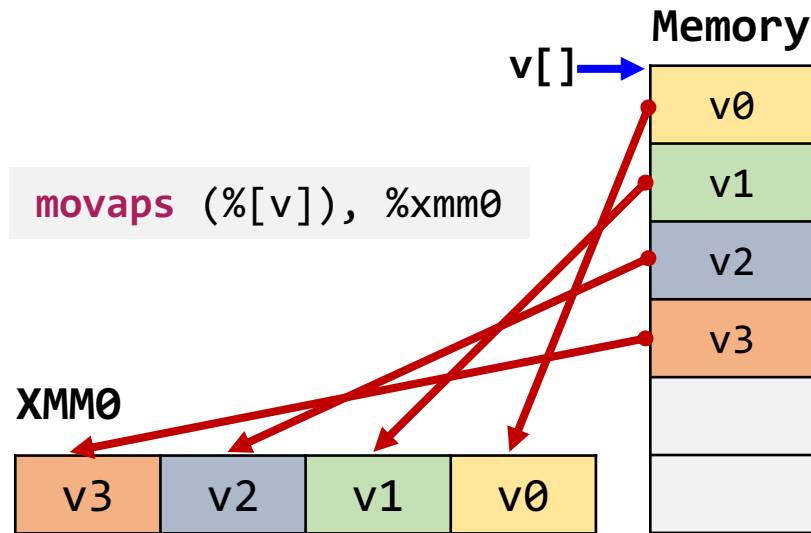
## ■ Поразрядные логические операции

- 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 <xmmintrin.h>    /* SSE, нужен также mmintrin.h */
#include <emmintrin.h>    /* SSE2, нужен также xmmintrin.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(align(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.

**Technologies**

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

**Categories**

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

**Functions**

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

?

<code>__m256d _mm256_broadcast_pd (__m128d const * mem_addr)</code>	<code>vbroadcastf128</code>
<code>__m256 _mm256_broadcast_ps (__m128 const * mem_addr)</code>	<code>vbroadcastf128</code>
<code>__m256d _mm256_broadcast_sd (double const * mem_addr)</code>	<code>vbroadcastsd</code>
<code>__m128 _mm_broadcast_ss (float const * mem_addr)</code>	<code>vbroadcastss</code>
<code>__m256 _mm256_broadcast_ss (float const * mem_addr)</code>	<code>vbroadcastss</code>
<code>__m256i _mm256_lddqu_si256 (__m256i const * mem_addr)</code>	<code>vlddqu</code>
<code>__m256d _mm256_load_pd (double const * mem_addr)</code>	<code>vmovapd</code>
<code>__m256 _mm256_load_ps (float const * mem_addr)</code>	<code>vmovaps</code>
<code>__m256i _mm256_load_si256 (__m256i const * mem_addr)</code>	<code>vmovdqa</code>
<code>__m256d _mm256_loadu_pd (double const * mem_addr)</code>	<code>vmovupd</code>
<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>

# 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
__m128 _mm_add_ss(__m128 a, __m128 b)	Addition	ADDSS
_mm_add_ps	Addition	ADDPS
_mm_sub_ss	Subtraction	SUBSS
_mm_sub_ps	Subtraction	SUBPS
_mm_mul_ss	Multiplication	MULSS
_mm_mul_ps	Multiplication	MULPS
_mm_div_ss	Division	DIVSS
_mm_div_ps	Division	DIVPS
_mm_sqrt_ss	Squared Root	SQRTSS
_mm_sqrt_ps	Squared Root	SQRTPS
_mm_rcp_ss	Reciprocal	RCPSS
_mm_rcp_ps	Reciprocal	RCPPS
_mm_rsqrt_ss	Reciprocal Squared Root	RSQRTSS
_mm_rsqrt_ps	Reciprocal Squared Root	RSQRTPS
_mm_min_ss	Computes Minimum	MINSS
_mm_min_ps	Computes Minimum	MINPS
_mm_max_ss	Computes Maximum	MAXSS
_mm_max_ps	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

# 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;
```

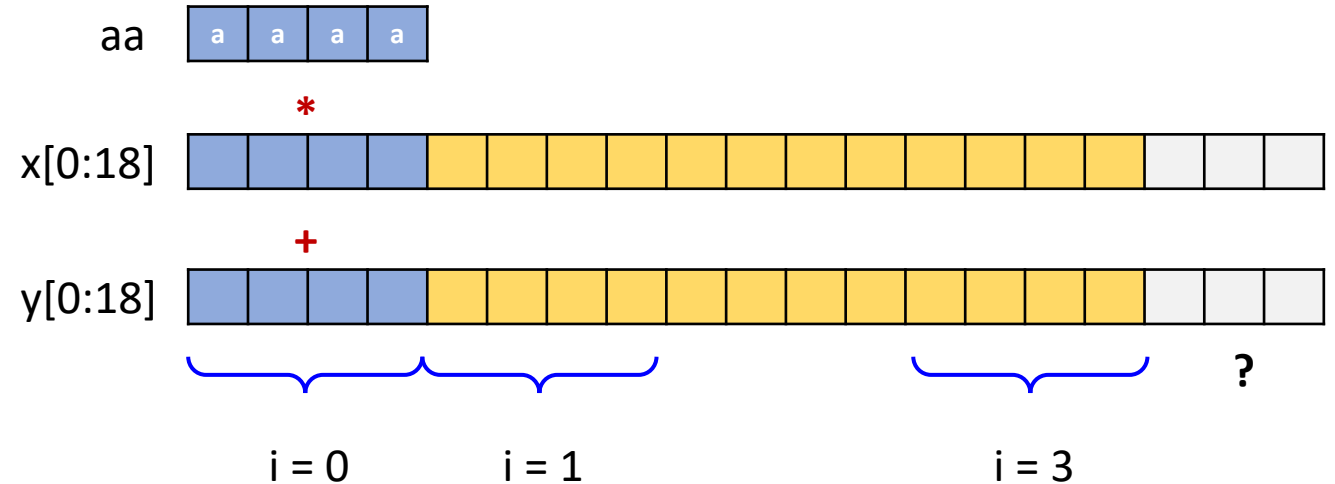
```
    }
```

```
    double t = wtime();
```

```
    saxpy_sse(x, y, a, n);
```

```
    t = wtime() - t;
```

```
}
```

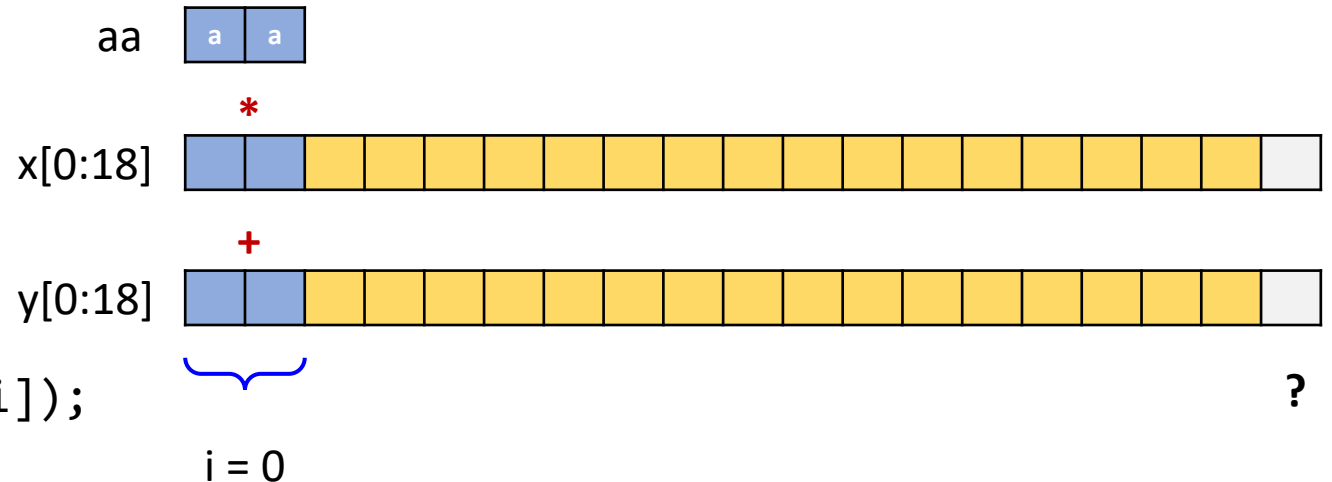


```
# Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
$ ./saxpy
SAXPY (y[i] = a * x[i] + y[i]; n = 1000000)
Elapsed time (scalar): 0.001571 sec.
Elapsed time (vectorized): 0.000835 sec.
Speedup: 1.88
```

# 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]);
    }
}
```



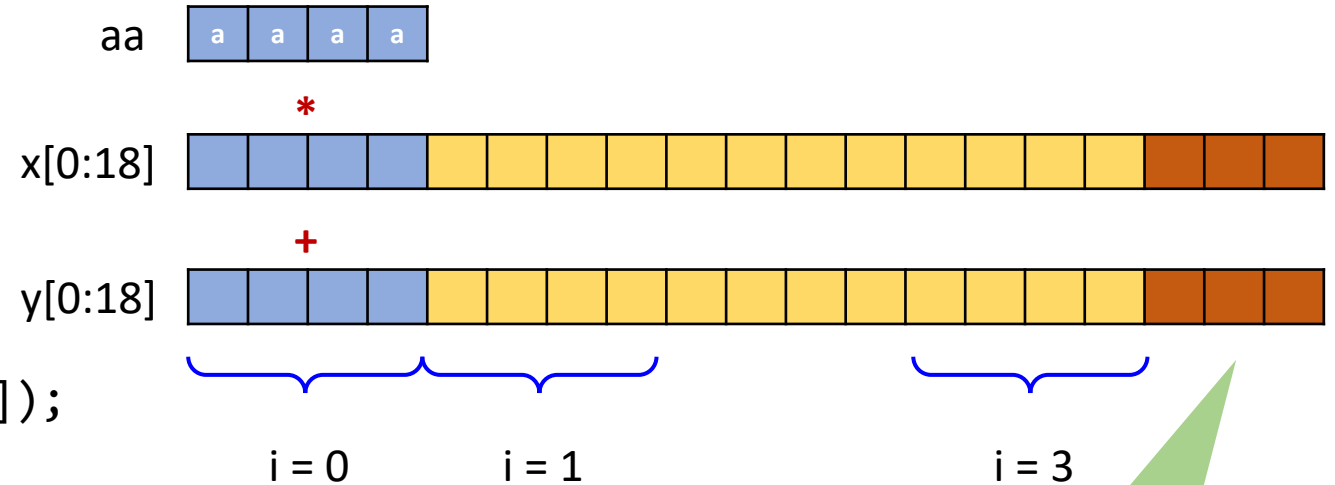
```
# Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
$ ./daxpy
daxpy (y[i] = a * x[i] + y[i]; n = 1000000)
Elapsed time (scalar): 0.002343 sec.
Elapsed time (vectorized): 0.001728 sec.
Speedup: 1.36
```

# 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];
}
```



Скалярная часть

# 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];
```

```
}
```

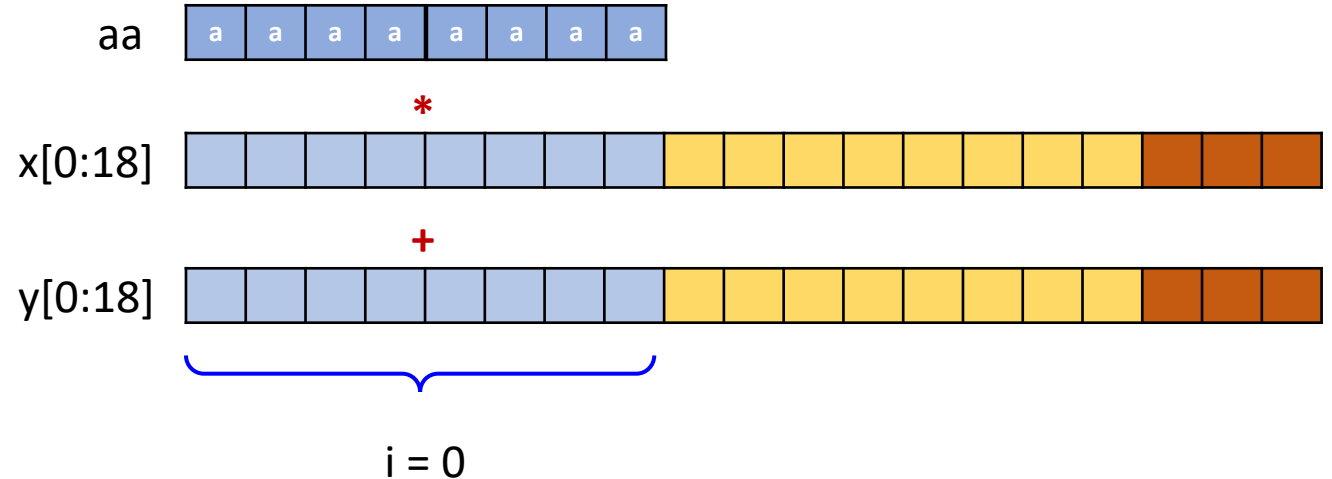
```
double run_vectorized()
```

```
{
```

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

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

```
    ...
```



# 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]);
    }
}
```